Thomas Wies (Ed.)

ARCoSS
LNCS 13990

Programming Languages and Systems

32nd European Symposium on Programming, ESOP 2023
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2023
Paris, France, April 22–27, 2023
Proceedings

ETAPS
EUROPEAN JOINT CONFERENCES ON
THEORY & PRACTICE OF SOFTWARE

Springer

OPEN ACCESS

Thomas Wies
Editor

Programming Languages and Systems

32nd European Symposium on Programming, ESOP 2023
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2023
Paris, France, April 22–27, 2023
Proceedings

Springer

Editor
Thomas Wies
New York University
New York, NY, USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-30043-1 ISBN 978-3-031-30044-8 (eBook)
https://doi.org/10.1007/978-3-031-30044-8

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

ETAPS Foreword

Welcome to the 26th ETAPS! ETAPS 2023 took place in Paris, the beautiful capital of France. ETAPS 2023 was the 26th 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 2023 received 361 submissions in total, 124 of which were accepted, yielding an overall acceptance rate of 34.3%. 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 2023 featured the unifying invited speakers Véronique Cortier (CNRS, LORIA laboratory, France) and Thomas A. Henzinger (Institute of Science and Technology, Austria) and the conference-specific invited speakers Mooly Sagiv (Tel Aviv University, Israel) for ESOP and Sven Apel (Saarland University, Germany) for FASE. Invited tutorials were provided by Ana-Lucia Varbanescu (University of Twente and University of Amsterdam, The Netherlands) on heterogeneous computing and Joost-Pieter Katoen (RWTH Aachen, Germany and University of Twente, The Netherlands) on probabilistic programming.

As part of the programme we had the second edition of TOOLympics, an event to celebrate the achievements of the various competitions or comparative evaluations in the field of ETAPS.

ETAPS 2023 was organized jointly by Sorbonne Université and Université Sorbonne Paris Nord. Sorbonne Université (SU) is a multidisciplinary, research-intensive and worldclass academic institution. It was created in 2018 as the merge of two first-class research-intensive universities, UPMC (Université Pierre and Marie Curie) and Paris-Sorbonne. SU has three faculties: humanities, medicine, and 55,600 students (4,700 PhD students; 10,200 international students), 6,400 teachers, professor-researchers and 3,600 administrative and technical staff members. Université Sorbonne Paris Nord is one of the thirteen universities that succeeded the University of Paris in 1968. It is a major teaching and research center located in the north of Paris. It has five campuses, spread over the two departments of Seine-Saint-Denis and Val d’Oise: Villetaneuse, Bobigny, Saint-Denis, the Plaine Saint-Denis and Argenteuil. The university has more than 25,000 students in different fields, such as health, medicine, languages, humanities, and science. The local organization team consisted of Fabrice Kordon (general co-chair), Laure Petrucci (general co-chair), Benedikt Bollig (workshops), Stefan Haar (workshops), Étienne André (proceedings and tutorials), Céline Ghibaudo (sponsoring), Denis Poitrenaud (web), Stefan Schwoon (web), Benoît Barbot (publicity), Nathalie Sznajder (publicity), Anne-Marie Reytier (communication), Hélène Pétridis (finance) and Véronique Criart (finance).

ETAPS 2023 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), EASST (European Association of Software Science and Technology), Lip6 (Laboratoire d'Informatique de Paris 6), LIPN (Laboratoire d'informatique de Paris Nord), Sorbonne Université, Université Sorbonne Paris Nord, CNRS (Centre national de la recherche scientifique), CEA (Commissariat à l'énergie atomique et aux énergies alternatives), LMF (Laboratoire méthodes formelles), and Inria (Institut national de recherche en informatique et en automatique).

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 Holger Hermanns (Saarbrücken), Marieke Huisman (Twente, chair), Jan Kofroň (Prague), Barbara König (Duisburg), Thomas Noll (Aachen), Caterina Urban (Inria), Jan Křetínský (Munich), and Lenore Zuck (Chicago).

Other members of the steering committee are: Dirk Beyer (Munich), Luís Caires (Lisboa), Ana Cavalcanti (York), Bernd Finkbeiner (Saarland), Reiko Heckel (Leicester), Joost-Pieter Katoen (Aachen and Twente), Naoki Kobayashi (Tokyo), Fabrice Kordon (Paris), Laura Kovács (Vienna), Orna Kupferman (Jerusalem), Leen Lambers (Cottbus), Tiziana Margaria (Limerick), Andrzej Murawski (Oxford), Laure Petrucci (Paris), Elizabeth Polgreen (Edinburgh), Peter Ryan (Luxembourg), Sriram Sankaranarayanan (Boulder), Don Sannella (Edinburgh), Natasha Sharygina (Lugano), Pawel Sobocinski (Tallinn), Sebastián Uchitel (London and Buenos Aires), Andrzej Wasowski (Copenhagen), Stephanie Weirich (Pennsylvania), Thomas Wies (New York), 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-Verlag GmbH for their support. I hope you all enjoyed ETAPS 2023.

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

April 2023

Marieke Huisman
ETAPS SC Chair
ETAPS e.V. President

Preface

This volume contains the papers accepted at the 32nd European Symposium on Programming (ESOP 2023), held during April 22–27, 2023, in Paris, France. ESOP is one of the European Joint Conferences on Theory and Practice of Software (ETAPS); it is dedicated to fundamental issues in the specification, design, analysis, and implementation of programming languages and systems.

The 20 papers in this volume were selected from 55 submissions based on their originality and quality. One submission was desk rejected due to formatting issues. Each of the remaining submissions received at least three reviews. Authors were given the opportunity to respond to the initial reviews of their papers during the rebuttal period, December 6–8, 2022. Afterwards, the papers were discussed by the 30 Program Committee (PC) members and the 37 external reviewers. ESOP 2023 followed a double-blind review process. Roland Meyer kindly handled the two papers for which the PC Chair had conflicts of interest.

ESOP 2023 continued the artifact evaluation process established by ESOP 2022. For this edition, the evaluation was conducted by a joint Artifact Evaluation Committee (AEC) with FoSSaCS 2023. Authors of accepted papers were invited to submit artifacts, such as code, datasets, and mechanized proofs that supported the conclusions of their papers. The AEC members read the papers and explored the artifacts, assessing their quality and checking that they supported the authors' claims. The authors of seven of the accepted papers submitted artifacts, which were evaluated by 21 AEC members, with each artifact receiving at least three reviews. Authors of papers with accepted artifacts were assigned official EAPLS artifact evaluation badges, indicating that they have taken the extra time and have undergone the extra scrutiny to prepare a useful artifact. The ESOP 2023 AEC awarded Artifact Functional, Artifact (Functional and) Reusable, and Artifact Available badges. All submitted artifacts were deemed Functional and Available, and all but two were also found to be Reusable.

I sincerely thank everyone who contributed to the success of the conference. Foremost, my deep gratitude goes to the authors who submitted their works for review, providing the basis for an exciting conference program. I would like to thank the members of the ESOP 2023 Program Committee for their detailed and constructive reviews, and for their active participation in the online discussions. The external reviewers provided additional expertise that was often crucial to arrive at an informed decision. For this, they have my deepest gratitude. I also thank Niccolò Veltri and Sebastian Wolff for serving as co-chairs of the joint ESOP/FoSSaCS 2023 Artifact Evaluation Committee. It was an honor to work with all of you! Finally, I would like to thank all who contributed to the organization of ESOP 2023: the ESOP steering committee and its chairs Luis Caires and Peter Thiemann, as well as the ETAPS steering committee and its chair Marieke Huisman, who often provided helpful guidance and feedback.

April 2023 Thomas Wies

Organization

Program Committee

Parosh Aziz Abdulla	Uppsala University, Sweden
Elvira Albert	Universidad Complutense de Madrid, Spain
Timos Antonopoulos	Yale University, USA
Suguman Bansal	Georgia Institute of Technology, USA
Josh Berdine	Meta, UK
Annette Bieniusa	TU Kaiserslautern, Germany
Sandrine Blazy	University of Rennes 1 - IRISA, France
Johannes Borgström	Uppsala University, Sweden
Georgiana Caltais	University of Twente, The Netherlands
Ankush Das	Amazon, USA
Cezara Dragoi	Inria Paris, ENS, France
Michael Emmi	Amazon Web Services, USA
Simon Gay	University of Glasgow, UK
Silvia Ghilezan	University of Novi Sad, Mathematical Institute SASA, Serbia
Jan Hoffmann	Carnegie Mellon University, USA
Shachar Itzhaky	Technion, Israel
Benjamin Lucien Kaminski	Saarland University, Saarland Informatics Campus, Germany
Robbert Krebbers	Radboud University Nijmegen, The Netherlands
Viktor Kuncak	Ecole Polytechnique Fédérale de Lausanne, Switzerland
Roland Meyer	TU Braunschweig, Germany
David Monniaux	CNRS/VERIMAG, France
Andrei Popescu	University of Sheffield, UK
Jonathan Protzenko	Microsoft, USA
Jorge A. Pérez	University of Groningen, The Netherlands
Graeme Smith	University of Queensland, Australia
Ana Sokolova	University of Salzburg, Austria
Alexander J. Summers	University of British Columbia, Canada
Tachio Terauchi	Waseda University, Japan
Caterina Urban	Inria, France
Niki Vazou	IMDEA Software, Spain
Thomas Wies	New York University, USA

Additional Reviewers

Abuah, Chike
Aman, Bogdan
Anastasiadi, Elli
Barrière, Aurèle
Bovel, Matthieu
Chassot, Samuel
Denis, Xavier
DeYoung, Henry
Di Giorgio, Alessandro
Eilers, Marco
Frumin, Daniil
Genaim, Samir
Goel, Aman
Goldstein, Mark
Gordillo, Pablo
Greenman, Ben
Grosen, Jessie
Ho, Son
Isabel, Miguel
Jacobs, Jules
Jothimurugan, Kishor
Khyzha, Artem
Kuperberg, Denis
Lam, Kait
Li, Yao
Liquori, Luigi
Middelkoop, Adriaan
Miné, Antoine
Padovani, Luca
Pham, Long
Rodríguez Carbonell, Enric
Rémy, Didier
Saville, Philip
Stanford, Caleb
Stein, Dario
Veltri, Niccolò
Wang, Di

Contents

Logics for Extensional, Locally Complete Analysis via Domain Refinements [⋆]

Flavio Ascari(✉), Roberto Bruni, and Roberta Gori

Dipartimento di Informatica, Università di Pisa, Largo B. Pontecorvo 3, Pisa, Italy,
`flavio.ascari@phd.unipi.it`, {`roberto.bruni`,`roberta.gori`}`@unipi.it`

Abstract. Abstract interpretation is a framework to design sound static analyses by over-approximating the set of program behaviours. While over-approximations can prove correctness, they cannot witness incorrectness because false alarms may arise. An ideal, but uncommon, situation is completeness of the abstraction that can ensure no false alarm is introduced by the abstract interpreter. Local Completeness Logic is a proof system that can decide both correctness and incorrectness of a program: any provable triple $\vdash_A [P]\ \mathsf{c}\ [Q]$ in the logic implies completeness of an *intensional* abstraction of program c on input P and is such that Q can be used to decide (in)correctness. However, completeness itself is an *extensional* property of the function computed by the program, while the above intensional analysis depends on the way the program is written and therefore not all valid triples can be derived in the proof system. Our main contribution is the study of new inference rules which allow one to perform part of the intensional analysis in a more precise abstract domain, and then to transfer the result back to the coarser domain. With these new rules, all (extensionally) valid triples can be derived in the proof system, thus untying the set of provable properties from the way the program is written.

Keywords: Abstract interpretation, Completeness in abstract interpretation, Hoare logic, Abstract domain refinement, Extensionality

1 Introduction

Static program analysis has been widely used to help developers produce valid software. Among static analysis techniques, abstract interpretation [6,7] is a general formalism to define sound-by-construction over-approximations that has been successfully applied in many fields, such as model checking, security and optimization [8]. Static analyses are often defined as *over-approximations*, that is the analysis computes a superset of the behaviors. This leads to no false negatives, that is all issues of the software are identified by the analysis, but it can cause false alarms: an incorrect behavior may be an artifact of the analysis, added by the over-approximation. While the absence of false negatives allowed a wide applicability of abstract interpretation techniques, it also make tools less

⋆ Research supported by MIUR PRIN Project 201784YSZ5 *ASPRA–Analysis of Program Analyses*.

reliable to identify bugs. In fact, in many industrial applications any false alarm reported by the analysis to the developers diminishes its credibility, making it less effective in practice. This argument has recently led to the development of a logic of under-approximations, called incorrectness logic [16,17].

The Problem. In abstract interpretation, an ideal situation is *completeness.* Given an *expressible* specification, that is, one represented exactly in the abstract domain, a complete abstraction reports no false alarms. In its most widespread formulation [7], completeness is a *global* property: a program c is complete in the abstraction A if a condition holds for all possible inputs. Let C be the concrete domain and $[\![\mathsf{c}]\!] : C \to C$ be the (collecting) denotational semantics of c. Given an abstract domain A, a concretization function $\gamma : A \to C$ and an abstraction function $\alpha : C \to A$, an abstract interpreter $[\![\mathsf{c}]\!]^\sharp_A : A \to A$ is complete in A if for all possible inputs P we have $[\![\mathsf{c}]\!]^\sharp_A \alpha(P) = \alpha([\![\mathsf{c}]\!] P)$. Unfortunately, because of universal quantification over the possible inputs, this condition is difficult to meet in practice. Moreover, in most cases completeness is checked on an intensional abstraction of $[\![\mathsf{c}]\!]$ computed inductively on the syntax, through inductive reasoning by an abstract interpreter $[\![\mathsf{c}]\!]^\sharp_A$ making completeness an *intensional* property dependent on the program syntax [10]. However, in principle completeness is an *extensional* property, that only depends on the best correct abstraction $[\![\mathsf{c}]\!]^A$ of $[\![\mathsf{c}]\!]$ in A, defined by $[\![\mathsf{c}]\!]^A \triangleq \alpha[\![\mathsf{c}]\!]\gamma$. We sum up what we may call intensional (on the left) and extensional (on the right) completeness in the following equations:

$$[\![\mathsf{c}]\!]^\sharp_A \alpha = \alpha [\![\mathsf{c}]\!] \qquad\qquad [\![\mathsf{c}]\!]^A \alpha = \alpha [\![\mathsf{c}]\!] \gamma \alpha = \alpha [\![\mathsf{c}]\!] \tag{1}$$

We show the difference between $[\![\mathsf{c}]\!]^A$ and $[\![\mathsf{c}]\!]^\sharp_A$ in the following example.

Example 1 (Extensional and intensional properties). Consider the concrete domain of sets of integers and the abstract domain of signs:

Sign

$$\begin{tikzcd}
& \mathbb{Z} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\mathbb{Z}_{\leq 0} \arrow[d, dash] \arrow[dr, dash] & \mathbb{Z}_{\neq 0} \arrow[dl, dash] \arrow[dr, dash] & \mathbb{Z}_{\geq 0} \arrow[dl, dash] \arrow[d, dash] \\
\mathbb{Z}_{<0} \arrow[dr, dash] & \mathbb{Z}_{=0} \arrow[d, dash] & \mathbb{Z}_{>0} \arrow[dl, dash] \\
& \varnothing &
\end{tikzcd}$$

The meaning of the abstract elements of Sign is to represent concrete values that satisfy the respective property. So for instance, denoting with the function γ the "meaning" of an abstract element, we have $\gamma(\mathbb{Z}_{<0}) = \{n \in \mathbb{Z} \,|\, n < 0\}$. Conversely, α "abstracts" a concrete set of values to the least abstract property describing it, for instance $\alpha(\{0; 1; 100\}) = \mathbb{Z}_{\geq 0}$.

Consider the simple program fragment $\mathsf{c} \triangleq$ `x := x + 1; x := x - 1`. Its denotational semantics $[\![\mathsf{c}]\!]$ is the identity function $\mathrm{id}_\mathbb{Z}$, so its best correct abstraction is the abstract identity $\mathrm{id}_{\mathsf{Sign}} = \alpha\ \mathrm{id}_\mathbb{Z}\ \gamma$. This is an *extensional* property of the program because it only depends on the function it computes, i.e., its denotational semantics. However, an analyzer does not know the semantics of c, so it has to analyze the program syntactically, breaking it down in elementary pieces and gluing the results together. So for instance, starting from the concrete point $P = \{1\}$ the analysis first abstracts it to the property $\alpha(P) = \mathbb{Z}_{>0}$, then it computes

$$\begin{aligned}\llbracket \mathsf{c} \rrbracket^\sharp_{\mathsf{Sign}}(\mathbb{Z}_{>0}) &= \llbracket \mathtt{x\ :=\ x\ -\ 1} \rrbracket^\sharp_{\mathsf{Sign}} \llbracket \mathtt{x\ :=\ x\ +\ 1} \rrbracket^\sharp_{\mathsf{Sign}}(\mathbb{Z}_{>0}) \\ &= \llbracket \mathtt{x\ :=\ x\ -\ 1} \rrbracket^\sharp_{\mathsf{Sign}}(\mathbb{Z}_{>0}) = \mathbb{Z}_{\geq 0}.\end{aligned}$$

Analogous calculations for all properties in Sign yields the abstraction

$$\llbracket \mathsf{c} \rrbracket^\sharp_{\mathsf{Sign}}(a) = \begin{cases} \bot & \text{if } a = \bot \\ \mathbb{Z}_{\geq 0} & \text{if } a \in \{ \boxed{\mathbb{Z}_{=0}}, \boxed{\mathbb{Z}_{>0}}, \mathbb{Z}_{\geq 0}\} \\ \mathbb{Z}_{<0} & \text{if } a = \mathbb{Z}_{<0} \\ \top & \text{if } a \in \{ \boxed{\mathbb{Z}_{\leq 0}}, \boxed{\mathbb{Z}_{\neq 0}}, \top\} \end{cases}$$

that, albeit sound, is less precise than $\mathrm{id}_{\mathsf{Sign}}$ (we highlight with a gray background all inputs on which $\llbracket \mathsf{c} \rrbracket^\sharp_{\mathsf{Sign}}$ loses accuracy). If instead the program were written as $\mathsf{c}' \triangleq \mathtt{skip}$, the analysis in Sign would yield the best correct abstraction $\llbracket \mathsf{c}' \rrbracket^\sharp_{\mathsf{Sign}} = \mathrm{id}_{\mathsf{Sign}}$. Therefore, the abstraction depends on how the program is written and not only on its semantics: it is what it is called an *intensional* property (see e.g. [1] for more about intensional and extensional abstract properties). $\square$

To overcome the former limitation of "global" completeness, the concept of *local* completeness [2] has been recently proposed that is related to some specific input. While this condition is much more common in practice, it is also much more complex to prove. In order to do so, the authors of [2] introduce a Local Completeness Logic parametric with respect to an abstraction A (LCL_A for short), that is able to prove triples $\vdash_A [P]\ \mathsf{c}\ [Q]$ with the following meaning

1. Q is an under-approximation of the concrete semantics $\llbracket \mathsf{c} \rrbracket P$,
2. Q and $\llbracket \mathsf{c} \rrbracket P$ have the same over-approximation in A,
3. A is locally complete for the intensional abstraction $\llbracket \mathsf{c} \rrbracket^\sharp_A$ on input P.

The important consequence of the previous points is the fact that a triple in LCL_A is able to prove *both correctness and incorrectness* of a program with respect to a specification *Spec* expressible in A. By point (2), if the abstract analysis reports no errors in Q then there are none because of the over-approximation. However, if the analysis does report an issue, this must be present in the abstraction of $\llbracket \mathsf{c} \rrbracket P$ as well, that is the same as the abstraction of Q: this means that Q contains a witness of the violation of *Spec*, and this witness must be in $\llbracket \mathsf{c} \rrbracket P$ because of the under-approximation ensured by point (1). While local completeness of point (3) is a key property to prove point (1-2), it would be enough to guarantee that (3) holds for the extensional best correct approximation $\llbracket \mathsf{c} \rrbracket^A$ of $\llbracket \mathsf{c} \rrbracket$ rather than for the intensional abstract interpreter $\llbracket \mathsf{c} \rrbracket^\sharp_A$: this suggests that it is possible to weaken the hypothesis (3) in order to make the proof system able to derive more valid triples.

Main Contributions. Building on the proof system of LCL_A, we add new rules to relax point (3) to local completeness of the extensional abstraction $[\![\mathsf{c}]\!]^A$. This way, while the proof system itself remains intensional as it deduces program properties by working inductively on the syntax, the information it produces is more precise. Specifically, since the property associated with triples is extensional no precision is lost because of the intensional abstract interpreter, and in the end allows us to prove more triples. In order to achieve this goal, we introduce new rules to *dynamically refine the abstract domain* during the analysis. While in general an analysis in a more concrete domain is more precise, LCL_A requires local completeness, which is not necessarily preserved by domain refinement [11]. For instance, a common way to combine two different abstract domains is their reduced product [7], but it is not always the case that the analysis in the reduced product is (locally) complete, even when it is such in the two domains.

To preserve local completeness, we introduce several rules for domain refinement in LCL_A and compare their expressiveness and usability. All of them provide extensional guarantees, in the sense that point (3) is replaced with local completeness of the best correct abstraction $[\![\mathsf{c}]\!]^A$ on input P. The first one is called (refine-ext). LCL_A extended with (refine-ext) turns out to be *logically complete*: any triple satisfying the above conditions (1–3) can be proved in our proof system. This is a theoretical improvement with respect to LCL_A, that instead was intrinsically incomplete as a logic, i.e., for all abstractions A there exists a sound triple that cannot be proved. While (refine-ext) is theoretically interesting, one of its hypothesis is unfeasible to check in practice. To improve applicability, we propose two derived rules, (refine-int) and (refine-pre), whose premises can be checked effectively and imply the hypotheses of the more general (refine-ext). Surprisingly, it turns out that (refine-int) enjoys a logical completeness result too, while (refine-pre) is strictly weaker (in terms of strength of the logic, see Example 6). Despite this, the latter is much simpler and preferable to use in practice whenever possible (see Example 5), while the former can be used in more situations and is at times the best choice.

We present a pictorial comparison among the expressiveness of the various proof systems in Fig. 1. Each node represent the proof system LCL_A extended with one rule (the bottom one being plain LCL_A). An arrow in the picture means a more powerful proof system, i.e., a proof system that can prove more triples, with its label pointing out the result justifying the claim. The two arrows between the two topmost nodes are because the two proof systems are logically equivalent, i.e., they can prove the same triples.

Structure of the paper. In Section 2 we explain the notation used in the paper and recall the basics of abstract interpretation. In Section 3 we present LCL_A, mostly summarizing the content of [2], with a focus on what is used in the following sections. In Section 4 we present and compare our new rules to refine the abstract domain, namely (refine-ext) and the two derived rules (refine-int) and (refine-pre). We conclude in Section 5. Some proofs and technical examples are in Appendix A.

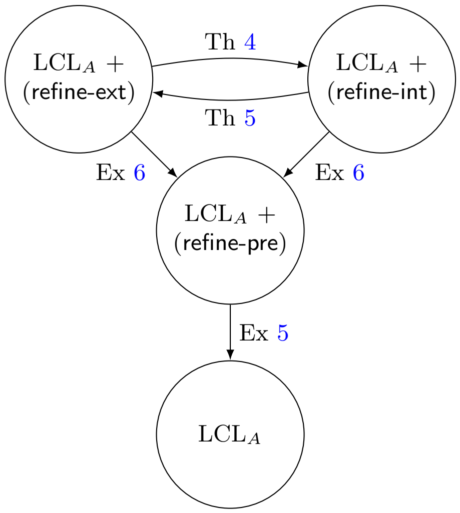


Fig. 1: Relations between the new proof systems

2 Background

Notation. We write $\mathcal{P}(S)$ for the powerset of S and $\mathrm{id}_S : S \to S$ for the identity function on a set S, with subscripts omitted when obvious from the context. If $f : S \to T$ is a function, we overload the symbol f to denote also its lifting $f : \mathcal{P}(S) \to \mathcal{P}(T)$ defined as $f(X) = \{f(x) \,|\, x \in X\}$ for any $X \subseteq S$. Given two functions $f : S \to T$ and $g : T \to V$ we denote their composition as $g \circ f$ or simply gf. For a function $f : S \to S$, we denote $f^n : S \to S$ the composition of f with itself n times, i.e. $f^0 = \mathrm{id}_S$ and $f^{n+1} = f \circ f^n$.

In ordered structures, such as posets and lattices, with carrier set C, we denote the ordering with $\leq_C$, least upper bounds (lubs) with $\sqcup_C$, greatest lower bounds (glbs) with $\sqcap_C$, least element with $\bot_C$ and greatest element with $\top_C$. For all these, we omit the subscript when evident from the context. Any powerset is a complete lattice ordered by set inclusion. In this case, we use standard symbols $\subseteq$, $\cup$, etc. Given a poset T and two functions $f, g : S \to T$, the notation $f \leq g$ means that, for all $s \in S$, $f(s) \leq_T g(s)$. A function f between complete lattices is additive (resp. co-additive) if it preserves arbitrary lubs (resp. glbs).

2.1 Abstract Interpretation

Abstract interpretation [6,7,5] is a general framework to define static analyses that are sound by construction. The main idea is to approximate the program semantics on some abstract domain A instead of working on the concrete domain C. The main tool used to study abstract interpretations are Galois connections. Given two complete lattices C and A, a pair of monotone functions $\alpha : C \to A$

and $\gamma : A \to C$ define a Galois connection (GC) when

$$\forall c \in C, a \in A. \quad \alpha(c) \leq_A a \iff c \leq_C \gamma(a).$$

We call C and A the concrete and the abstract domain respectively, α the abstraction function and γ the concretization function. The functions α and γ are also called adjoints. For any GC, it holds $\mathrm{id}_C \leq \gamma\alpha$, $\alpha\gamma \leq \mathrm{id}_A$, γ is co-additive and α is additive. A concrete value $c \in C$ is called *expressible* in A if $\gamma\alpha(c) = c$. We only consider GCs in which $\alpha\gamma = \mathrm{id}_A$, called Galois insertions (GIs). In a GI α is onto and γ is injective. A GI is said to be trivial if A is isomorphic to the concrete domain or if it is the singleton $\{\top_A\}$.

We overload the symbol A to denote also the function $\gamma\alpha : C \to C$: this is always a closure operator, that is a monotone, increasing (i.e. $c \leq A(c)$ for all c) and idempotent function. In the following, we use closure operators as much as possible to simplify the notation. Particularly, they are useful to denote domain refinements, as exemplified in the next paragraph. Note that they are still very expressive because γ is injective: for instance $A(c) = A(c')$ if and only if $\alpha(c) = \alpha(c')$. Nonetheless, the use of closure operators is only a matter of notation and it is always possible to rewrite them using the adjoints.

We use $\mathrm{Abs}(C)$ to denote the set of abstract domains over C, and we write $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ when we need to make the two maps α and γ explicit (we omit them when not needed). Given two abstract domains $A_{\alpha,\gamma}, A'_{\alpha',\gamma'} \in \mathrm{Abs}(C)$ over C, we say A' is a *refinement* of A, written $A' \preceq A$, when $\gamma(A) \subseteq \gamma'(A')$. When this happens, the abstract domain A' is more expressive than A, and in particular for all concrete elements $c \in C$ the inequality $A'(c) \leq_C A(c)$ holds.

Abstracting Functions. Given a monotone function $f : C \to C$ and an abstract domain $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$, a function $f^\sharp : A \to A$ is a sound approximation (or abstraction) of f if $\alpha f \leq f^\sharp \alpha$. Its *best correct approximation* (bca) is $f^A = \alpha f \gamma$, and it is the most precise of all the sound approximations of f: a function $f^\sharp$ is a sound approximation of f if and only if $f^A \leq f^\sharp$.

A sound abstraction $f^\sharp$ of f is *complete* if $\alpha f = f^\sharp \alpha$. It turns out that there exists a complete abstraction $f^\sharp$ if and only if the bca f^A is complete. If this is the case, we say that the abstract domain A is complete for f and denote it with $\mathbb{C}^A(f)$. Intuitively, completeness means that the abstract function $f^\sharp$ is as precise as possible in the given abstract domain A, and in program analysis this allows to have greater confidence in the alarms raised. We remark that A is complete for f if and only if $\alpha f = f^A \alpha = \alpha f \gamma \alpha$. Since γ is injective, this is true if and only if $\gamma\alpha f = \gamma\alpha f \gamma\alpha$, so that we define the (global) completeness property $\mathbb{C}^A(f)$ as follows:

$$\mathbb{C}^A(f) \iff Af = AfA.$$

2.2 Regular Commands.

Following [2] (see also [16]) we consider a language of *regular commands*:

$$\mathsf{Reg} \ni \mathsf{r} ::= \mathsf{e} \mid \mathsf{r};\mathsf{r} \mid \mathsf{r} \oplus \mathsf{r} \mid \mathsf{r}^*$$

This is a general language and can be instantiated differently changing the set Exp of basic transfer expressions e. These determines the kind of operations allowed in the language, and in our examples we assume to have deterministic assignments and boolean guards. Using standard definitions for arithmetic and boolean expressions $\mathsf{a} \in \mathrm{AExp}$ and $\mathsf{b} \in \mathrm{BExp}$, we consider

$$\mathsf{Exp} \ni \mathsf{e} ::= \mathtt{skip} \mid \mathtt{x\ :=\ a} \mid \mathtt{b?}$$

$\mathtt{skip}$ does nothing, $\mathtt{x\ :=\ a}$ is a standard deterministic assignment. The semantics of $\mathtt{b?}$ is that of an "assume" statement: if its input satisfies b it does nothing, otherwise it diverges. The term $\mathsf{r}; \mathsf{r}$ represent the usual sequential composition, and $\mathsf{r} \oplus \mathsf{r}$ is nondeterministic choice. The Kleene star r^* denote a nondeterministic iteration, where r can be executed any number of time (possibly 0) before exiting. It can be thought as the solution of the recursive equation $\mathsf{r}^* \equiv \mathtt{skip} \oplus (\mathsf{r}; \mathsf{r}^*)$. We write r^n to denote sequential composition of r with itself n times, analogously to how we use f^n for function composition.

This formulation can accommodate for a standard imperative programming language [18] defining $\mathtt{if}$ and $\mathtt{while}$ statements as

$$\mathtt{if\ (b)\ then\ c_1\ else\ c_2} \triangleq (\mathtt{b?;\ c_1}) \oplus ((\neg \mathtt{b})\mathtt{?;\ c_2})$$
$$\mathtt{while\ (b)\ do\ c} \triangleq (\mathtt{b?;\ c})^*; (\neg \mathtt{b})\mathtt{?}$$

Concrete semantics. We assume the semantics $(\!|\cdot|\!) : \mathsf{Exp} \to C \to C$ of basic transfer expressions on a complete lattice C to be additive. We believe this assumption not to be restrictive, and is always satisfied in collecting semantics. For our instantiation of Exp, we consider a finite set of variables Var, then the set of stores $\Sigma = \mathrm{Var} \to \mathbb{Z}$ that are (total) functions σ from Var to integers. The complete lattice C is then defined simply as $\mathcal{P}(\Sigma)$ with the usual poset structure given by set inclusion. Given a store $\sigma \in \Sigma$, store update $\sigma[x \mapsto v]$ is defined as usual for $x \in \mathrm{Var}$ and $v \in \mathbb{Z}$. We consider standard, inductively defined semantics $(\!|\cdot|\!)$ for arithmetic and boolean expressions. The concrete semantics of regular commands $[\![\cdot]\!] : \mathsf{Reg} \to C \to C$ is defined inductively as in Fig. 2a, where the semantics of basic transfer expressions $\mathsf{e} \in \mathsf{Exp}$ is defined as follows:

$$(\!|\mathtt{skip}|\!)S \triangleq S$$
$$(\!|\mathtt{x\ :=\ a}|\!)S \triangleq \{\sigma[x \mapsto (\!|\mathsf{a}|\!)\sigma] \mid \sigma \in S\}$$
$$(\!|\mathtt{b?}|\!)S \triangleq \{\sigma \in S \mid (\!|\mathsf{b}|\!)\sigma = \mathtt{tt}\}$$

Abstract Semantics. The (compositional) abstract semantics of regular commands $[\![\cdot]\!]^\sharp_A : \mathsf{Reg} \to A \to A$ on an abstract domain $A \in \mathrm{Abs}(C)$ is defined inductively as in Fig. 2b. As common for abstract interpreters, we assume the analyser knows the best correct abstraction of expression and thus is able to compute $[\![\mathsf{e}]\!]^A$. A straightforward proof by structural induction shows that the abstract semantics is sound w.r.t. $[\![\mathsf{r}]\!]$ (i.e., $\alpha[\![\mathsf{r}]\!] \leq [\![\mathsf{r}]\!]^\sharp_A \alpha$) and monotone. However, in general it is less precise than the bca, i.e., $[\![\mathsf{r}]\!]^\sharp_A \neq [\![\mathsf{r}]\!]^A = \alpha[\![\mathsf{r}]\!]\gamma$.

$$\begin{aligned}
[\![\mathsf{e}]\!]c &\triangleq (\!|\mathsf{e}|\!)c \\
[\![\mathsf{r}_1; \mathsf{r}_2]\!]c &\triangleq [\![\mathsf{r}_2]\!][\![\mathsf{r}_1]\!](c) \\
[\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]c &\triangleq [\![\mathsf{r}_1]\!]c \sqcup_C [\![\mathsf{r}_2]\!]c \\
[\![\mathsf{r}^*]\!]c &\triangleq \bigsqcup_{n \geq 0} [\![\mathsf{r}]\!]^n c
\end{aligned}$$

(a) Concrete semantics

$$\begin{aligned}
[\![\mathsf{e}]\!]^\sharp_A a &\triangleq [\![\mathsf{e}]\!]^A a = \alpha(\!|\mathsf{e}|\!)\gamma(a) \\
[\![\mathsf{r}_1; \mathsf{r}_2]\!]^\sharp_A a &\triangleq [\![\mathsf{r}_2]\!]^\sharp_A [\![\mathsf{r}_1]\!]^\sharp_A (a) \\
[\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]^\sharp_A a &\triangleq [\![\mathsf{r}_1]\!]^\sharp_A a \sqcup_A [\![\mathsf{r}_2]\!]^\sharp_A a \\
[\![\mathsf{r}^*]\!]^\sharp_A a &\triangleq \bigsqcup_{n \geq 0} ([\![\mathsf{r}]\!]^\sharp_A)^n a
\end{aligned}$$

(b) Abstract semantics

Fig. 2: Concrete and abstract semantics of regular commands, side by side

Shorthands. Throughout the paper, we present some simple examples of program analysis. The programs discussed in the examples contain just one or two variables (usually `x` and `y`), so we denote their sets of stores just as $\Sigma = \mathbb{Z}$ or $\Sigma = \mathbb{Z}^2$. In these cases, the convention is that an element of $\mathbb{Z}$ is the value of the single variable in Var, and a pair $(n, m) \in \mathbb{Z}^2$ denote the store $\sigma(\mathtt{x}) = n$, $\sigma(\mathtt{y}) = m$. We also lift these conventions to sets of values in $\mathbb{Z}$ or $\mathbb{Z}^2$. At times, to improve readability, we use logical formulas such as $(y \in \{1; 2; 99\} \wedge x = y)$ possibly using intervals, like in $x \in [0; 5]$, to describe set of stores.

3 Local Completeness Logic

In this section we present the notion of local completeness and introduce the proof system LCL$_A$ (Local Completeness Logic on A) as was defined in [2].

For a generic program and abstract domain, global completeness is a too strong requirement: for conditionals to be complete the abstract domain should basically contain a complete sublattice of the concrete domain. For this reason, the weaker notion of *local* completeness can be more convenient in many cases.

Definition 1 (Local completeness, cf. [2]). *Let $f : C \to C$ be a concrete function, $c \in C$ a concrete point and $A \in$ Abs(C) and abstract domain for C. Then A is* locally complete *for f on c, written $\mathbb{C}^A_c(f)$, iff*

$$Af(c) = AfA(c).$$

A remarkable difference between global and local completeness is that, while the former can be proved compositionally irrespective of the input [10], the latter needs it. Consequently, to carry on a compositional proof of local completeness, information on the input to each subpart of the program is also required, i.e., all traversed states are important. However, local completeness enjoys an "abstract convexity" property, that is, local completeness on a concrete point c implies local completeness on any concrete point d between c and its abstraction $A(c)$. This observation has been exploited in the design of the proof system LCL$_A$. The system is able to prove triples $\vdash_A [P]\ \mathsf{r}\ [Q]$ ensuring that:

$$\frac{\mathbb{C}^A_P(\llbracket \mathsf{e} \rrbracket)}{\vdash_A [P]\ \mathsf{e}\ [\llbracket \mathsf{e} \rrbracket P]}\ (\mathsf{transfer}) \qquad \frac{P' \leq P \leq A(P') \quad \vdash_A [P']\ \mathsf{r}\ [Q'] \quad Q \leq Q' \leq A(Q)}{\vdash_A [P]\ \mathsf{r}\ [Q]}\ (\mathsf{relax})$$

$$\frac{\vdash_A [P]\ \mathsf{r}_1\ [R] \quad \vdash_A [R]\ \mathsf{r}_2\ [Q]}{\vdash_A [P]\ \mathsf{r}_1 ; \mathsf{r}_2\ [Q]}\ (\mathsf{seq}) \qquad \frac{\vdash_A [P]\ \mathsf{r}_1\ [Q_1] \quad \vdash_A [P]\ \mathsf{r}_2\ [Q_2]}{\vdash_A [P]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [Q_1 \vee Q_2]}\ (\mathsf{join})$$

$$\frac{\vdash_A [P]\ \mathsf{r}\ [R] \quad \vdash_A [P \vee R]\ \mathsf{r}^*\ [Q]}{\vdash_A [P]\ \mathsf{r}^*\ [Q]}\ (\mathsf{rec}) \qquad \frac{\vdash_A [P]\ \mathsf{r}\ [Q] \quad Q \leq A(P)}{\vdash_A [P]\ \mathsf{r}^*\ [P \vee Q]}\ (\mathsf{iterate})$$

Fig. 3: The proof system LCL_A.

1. Q is an under-approximation of the concrete semantics $\llbracket \mathsf{r} \rrbracket P$,
2. Q and $\llbracket \mathsf{r} \rrbracket P$ have the same over-approximation in A,
3. A is locally complete for $\llbracket \mathsf{r} \rrbracket$ on input P.

The second point means that, given a specification *Spec* expressible in A, any provable triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ either proves correctness of r with respect to *Spec* or expose some alerts in $Q \setminus \mathit{Spec}$. These in turns correspond to true ones because of the first point, as spelled out by Corollary 1 below.

The proof system is defined in Fig. 3. The crux of the proof system is to constrain the under-approximation Q to have the same abstraction of the concrete semantics $\llbracket \mathsf{r} \rrbracket P$, as for instance explicitly required in rule (relax). This, by the abstract convexity property mentioned above, means that local completeness of $\llbracket \mathsf{r} \rrbracket$ on the *under-approximation* P of the concrete store is enough to prove local completeness.

The three key properties (1–3) listed above are formalized by the following (intensional) soundness result:

Theorem 1 (Soundness, cf. [2]). *Let* $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$. *If* $\vdash_A [P]\ \mathsf{r}\ [Q]$ *then:*

1. $Q \leq \llbracket \mathsf{r} \rrbracket P$,
2. $\alpha(\llbracket \mathsf{r} \rrbracket P) = \alpha(Q)$,
3. $\llbracket \mathsf{r} \rrbracket^\sharp_A \alpha(P) = \alpha(Q)$.

As a consequence of this theorem, given a specification expressible in the abstract domain A, a provable triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ can determine both correctness and incorrectness of the program r:

Corollary 1 (Proofs of Verification, cf. [2]). *Let* $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ *and* $a \in A$. *If* $\vdash_A [P]\ \mathsf{r}\ [Q]$ *then*

$$\llbracket \mathsf{r} \rrbracket P \leq \gamma(a) \iff Q \leq \gamma(a).$$

The corollary is useful in program analysis and verification because, given a specification a expressible in A and a provable triple $\vdash_A [P]\ \mathsf{r}\ [Q]$, it allows to distinguish two cases.

– If $Q \subseteq \gamma(a)$, then we have also $\llbracket \mathsf{r} \rrbracket P \subseteq \gamma(a)$, so that the program is correct with respect to the specification.

– If $Q \not\subseteq \gamma(a)$, then also $[\![\mathsf{r}]\!]P \not\subseteq \gamma(a)$, that means $[\![\mathsf{r}]\!]P \setminus \gamma(a)$ is not empty and thus contains a true alert of the program. Moreover, since $Q \subseteq [\![\mathsf{r}]\!]P$ we have that $Q \setminus \gamma(a) \subseteq [\![\mathsf{r}]\!]P \setminus \gamma(a)$, so that already Q pinpoints some issues.

To better show how this work, we briefly introduce the following example (discussed also in [2] where it is possible to find all details of the derivation).

Example 2. Consider the concrete domain $C = \mathcal{P}(\mathbb{Z})$, the abstract domain Int of intervals, the precondition $P = \{1; 999\}$ and the command $\mathsf{r} \triangleq (\mathsf{r}_1 \oplus \mathsf{r}_2)^*$, where

$$\mathsf{r}_1 \triangleq \texttt{(x > 0)?; x := x - 1}$$
$$\mathsf{r}_2 \triangleq \texttt{(x < 1000)?; x := x + 1}$$

In LCL$_A$ it is possible to prove the triple $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [Q]$, whose postcondition is $Q = \{0; 2; 1000\}$. Consider the two specification $\mathit{Spec} = (x \leq 1000)$ and $\mathit{Spec}' = (x \geq 100)$. The triple is then able to prove correctness of Spec and incorrectness of Spec'. For the former, observe that $Q \subseteq \mathit{Spec}$. By Corollary 1 we then know $[\![\mathsf{r}]\!]P \subseteq \mathit{Spec}$, that is correctness. For the latter, Q exhibits two witnesses to the violation of Spec', that are $0, 2 \in Q \setminus \mathit{Spec}'$. By point (1) of soundness we then know that $0, 2 \in Q \subseteq [\![\mathsf{r}]\!]P$ are true alerts. □

Strictly speaking, the proof of Corollary 1 only relies on points (1-2) of Theorem 1. Point (3) is in turn needed to ensure the first two, but extensional completeness would suffice to this aim. This means that we can weaken the soundness theorem (logically speaking, that is we prove a stronger conclusion, so the theorem as an implication is weaker) while still preserving the validity of Corollary 1. To this end, we propose a new soundness result involving extensional completeness: the important difference is that in point (3) we use the best correct abstraction $[\![\mathsf{r}]\!]^A$ in place of the inductively defined $[\![\mathsf{r}]\!]^\sharp_A$. Since Theorem 1 involves $[\![\mathsf{r}]\!]^\sharp_A$, an *intensional* property of the program r that depends on how the program is written (see Example 1 or Example 1 in Section 5 of [13]), while the new statement we propose relies only on $[\![\mathsf{r}]\!]^A$, an *extensional* property of the computed function $[\![\mathsf{r}]\!]$ and not of r itself, for the rest of the paper we use the name *intensional soundness* for Theorem 1, and *extensional soundness* for the following Theorem 2.

Theorem 2 (Extensional soundness). *Let $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$. If $\vdash_A [P]\ \mathsf{r}\ [Q]$ then:*

1. *$Q \leq [\![\mathsf{r}]\!]P$,*
2. *$\alpha([\![\mathsf{r}]\!]P) = \alpha(Q)$,*
3. *$[\![r]\!]^A \alpha(P) = \alpha(Q)$.*

Lastly, we remark that the original LCL$_A$ is intrinsically logically incomplete ([2], cf. Theorem 5.12): for every non trivial abstraction A there exists a triple that is intensionally sound (satisfies points (1-3) of Theorem 1) but cannot be proved in LCL$_A$. We will discuss logical (in)completeness for our extensional framework in Section 4.1.

$$\frac{\vdash_{A'} [P]\ \mathsf{r}\ [Q] \quad A' \preceq A \quad A[\![\mathsf{r}]\!]^{A'}A(P) = A(Q)}{\vdash_A [P]\ \mathsf{r}\ [Q]}\ (\mathsf{refine\text{-}ext})$$

Fig. 4: Rule refine for LCL_A.

4 Refining Abstract Domain

LCL_A can prove a triple $[P]$ r $[Q]$ for some Q only when $[\![\mathsf{r}]\!]^\sharp_A$ is locally complete, that is $[\![\mathsf{r}]\!]^\sharp_A\alpha(P) = \alpha([\![\mathsf{r}]\!]P)$ (see Theorem 1). Since $[\![\mathsf{r}]\!]^\sharp_A$ is computed in a compositional way, the above condition strictly depends on how r is written: to prove the local completeness of $[\![\mathsf{r}]\!]^\sharp_A$, we need to prove that all its syntactic components are locally complete, that is an intensional property. However, the goal of the analysis is to study the behaviour of the function $[\![\mathsf{r}]\!]$, not how it is encoded by r. Hence, our aim is to enhance the original proof system in order to be able to handle triples where the extensional abstraction $[\![\mathsf{r}]\!]^A$ is proved to be locally complete w.r.t. the given input, that is $[\![\mathsf{r}]\!]^A\alpha(P) = \alpha([\![\mathsf{r}]\!]P)$. To this end, we extend the proof system with a new inference rule, that is shown in Fig. 4. It is named after "refine" because it allows to refine abstract domains A to some $A' \preceq A$ and "ext" since it involves the extensional bca $[\![\mathsf{r}]\!]^{A'}$ of $[\![\mathsf{r}]\!]$ in A' (to distinguish it from the rules we will introduce in Section 4.2).

Using (refine-ext) it is possible to construct a derivation that proves local completeness of portions of the whole program in a more precise abstract domain A' and then carries the result over to the global analysis in a coarser domain A. The only requirement for the application of the rule is that domain A' is chosen in such a way that $A[\![\mathsf{r}]\!]^{A'}A(P) = A(Q)$ is satisfied.

Formally, given the two abstract domains $A_{\alpha,\gamma}, A'_{\alpha',\gamma'} \in \text{Abs}(C)$, this last premise of rule (refine-ext) should be written as $\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha'A(P) = \alpha(Q)$ to match function domains and codomains. However we prefer the more concise, albeit a little imprecise, notation used in Fig. 4. That writing is justified by the following intuitive argument: since $A' \preceq A$ we can consider with a slight abuse of notation (seeing abstract domains as closures) $A \subseteq A' \subseteq C$, so that for any element $a \in A \subseteq C$ we have $\gamma(a) = \gamma'(a) = a$ and for any $c \in C$ we have $\alpha'A(c) = A(c)$. With these, it follows that

$$\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha'A(P) = \alpha[\![\mathsf{r}]\!]^{A'}A(P) = A[\![\mathsf{r}]\!]^{A'}A(P).$$

With rule (refine-ext) we cannot prove intensional soundness (Theorem 1): since this rule allows to perform part of the analysis in a more concrete domain A', we do not get any information on $[\![\mathsf{r}]\!]^\sharp_A$. However, we can prove extensional soundness (Theorem 2) and get all the benefits of Corollary 1.

Theorem 3 (Extensional soundness of (refine-ext)). *The proof system in Fig. 3 with the addition of rule* (refine-ext) *(see Fig. 4) is extensionally sound (cf. Theorem 2).*

We also remark that a rule like (refine-ext), that allows to carry on part of the proof in a different abstract domain, cannot come unconstrained. We present an example showing that a similar inference rule only requiring the triple $[P]\ \mathsf{r}\ [Q]$ to be provable in an abstract domain $A' \preceq A$ without any other constraint would be unsound.

Example 3. Consider the concrete domain $C = \mathcal{P}(\mathbb{Z})$ of integers, the point $P = \{-5; -1\}$, the abstract domain Sign of Example 1 and the program

$$\mathsf{r} \triangleq \texttt{x := x + 10}.$$

Then $C \preceq \mathsf{Sign}$ and we can prove $\vdash_C [P]\ \mathsf{r}\ [\{5; 9\}]$ applying (transfer) since all assignments are locally complete in the concrete domain. However, if $f = [\![\mathsf{r}]\!] = (\![\texttt{x := x + 10}]\!)$, it is not the case that $\mathbb{C}_P^{\mathsf{Sign}}(f)$: indeed

$$\mathsf{Sign}(f(\mathsf{Sign}(P))) = \mathsf{Sign}(f(\mathbb{Z}_{<0})) = \mathsf{Sign}(\{n \in \mathbb{Z} \mid n < 10\}) = \top$$

while

$$\mathsf{Sign}(f(P)) = \mathsf{Sign}(\{5, 9\}) = \mathbb{Z}_{>0}.$$

This means that a rule without any additional condition can prove a triple which is not locally complete, hence it is unsound. □

4.1 Logical Completeness

Among all the possible conditions that can be added to a rule like (refine-ext), we believe ours to be very general since, differently than the original LCL$_A$ proof system (see Section 5.2 of [2]), the introduction of (refine-ext) allows us to derive a logical completeness result, i.e. the ability to prove *any* triple satisfying the soundness properties guaranteed by the proof system.

However, to prove such a result, our extension need an additional rule to handle loops, just like the original LCL$_A$ and Incorrectness Logic [16]. The necessary infinitary rule, called (limit), allows the proof system to handle Kleene star, and is the same as LCL$_A$:

$$\frac{\forall n \in \mathbb{N}.\ \vdash_A [P_n]\ \mathsf{r}\ [P_{n+1}]}{\vdash_A [P_0]\ \mathsf{r}^*\ [\bigvee_{i \in \mathbb{N}} P_i]}\ (\mathsf{limit})$$

Theorem 4 (Logical completeness of (refine-ext)). *Consider the proof system of Fig. 3 with the addition of rules* (refine-ext) *and* (limit)*. If $Q \le [\![\mathsf{r}]\!]P$ and $[\![\mathsf{r}]\!]^A \alpha(P) = \alpha(Q)$ then $\vdash_A [P]\ \mathsf{r}\ [Q]$.*

The previous theorem proves the logical completeness of our proof system with respect to the property of extensional soundness. Indeed, if $Q \le [\![\mathsf{r}]\!]P$ and $[\![\mathsf{r}]\!]^A \alpha(P) = \alpha(Q)$ we also have:

$$\alpha(Q) \le \alpha([\![\mathsf{r}]\!]P) \le [\![\mathsf{r}]\!]^A \alpha(P) = \alpha(Q),$$

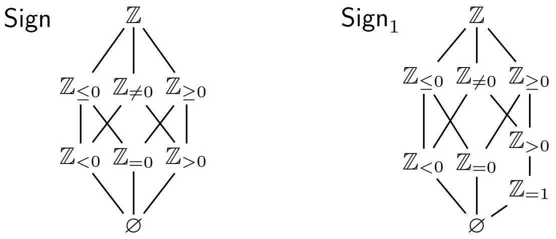
hence all three conditions of Theorem 2 are satisfied.

An interesting consequence of this result is the existence of a refinement A' in which it is possible to carry out the proof. In principle such a refinement could be the concrete domain C (as shown in the proof in Appendix A), that is not computable. However, it is worth nothing that for a sequential fragment (a portion of code without loops) the concrete domain can be actually used (for instance via first-order logic). This opens up the possibility, for instance, to infer a loop invariant on the body using C, and then prove it using an abstract domain. In Section 4.3 we discuss this issue further.

4.2 Derived Refinement Rules

The hypothesis $A[\![\mathsf{r}]\!]^{A'}A(P) = A(Q)$ is added to rule (refine-ext) in order to guarantee soundness: in general, the ability to prove a triple such as $[P]\ \mathsf{r}\ [Q]$ in a refined domain A' only gives information on $A[\![\mathsf{r}]\!]^{A'}A'(P)$ but not on $A[\![\mathsf{r}]\!]^{A'}A(P)$. In fact, the Example 4 shows that $A[\![\mathsf{r}]\!]^{A'}A'(P)$ and $A[\![\mathsf{r}]\!]^{A'}A(P)$ can be different.

Example 4. Consider the concrete domain $\mathcal{P}(\mathbb{Z})$, the abstract domain of signs $\mathsf{Sign}_{\alpha,\gamma} \in \mathrm{Abs}(\mathcal{P}(\mathbb{Z}))$ (introduced in Example 1) and its refinement Sign_1 below



For the command $\mathsf{r} \triangleq$ `x := x - 1` and the concrete point $P = \{1\}$ we have

$$\mathsf{Sign}[\![\mathsf{r}]\!]^{\mathsf{Sign}_1}\mathsf{Sign}_1(P) = \mathsf{Sign}[\![\mathsf{r}]\!]^{\mathsf{Sign}_1}(\mathbb{Z}_{=1}) = \mathbb{Z}_{=0}$$

but

$$\mathsf{Sign}[\![\mathsf{r}]\!]^{\mathsf{Sign}_1}\mathsf{Sign}(P) = \mathsf{Sign}[\![\mathsf{r}]\!]^{\mathsf{Sign}_1}(\mathbb{Z}_{>0}) = \mathbb{Z}_{\geq 0}. \qquad \square$$

Despite being necessary, the hypothesis of rule (refine-ext) cannot be checked in practice because the bca $[\![\mathsf{r}]\!]^{A'}$ of a composite command r is not known by the analyser. To mitigate this issue, we present two derived rules whose premises imply the premises of Rule (refine-ext), hence ensuring extensional soundness by means of Theorem 3.

The first rule we present replaces the requirement on the extensional bca $[\![\mathsf{r}]\!]^{A'}$ with requirements on the intensional compositional abstraction $[\![\mathsf{r}]\!]^{\sharp}_{A'}$ computed in A'. For this reason, we call this rule (refine-int).

Proposition 1. *The following rule* (refine-int) *is extensionally sound:*

$$\frac{\vdash_{A'} [P]\ \mathsf{r}\ [Q] \quad A' \preceq A \quad A[\![\mathsf{r}]\!]^{\sharp}_{A'}A(P) = A(Q)}{\vdash_{A} [P]\ \mathsf{r}\ [Q]}\ (\text{refine-int})$$

It is worth noting that now the condition on the compositional abstraction $[\![\mathsf{r}]\!]^\sharp_{A'}$ can easily be checked by the analyser, possibly alongside the analysis of r with LCL or using a stand-alone abstract interpreter. Moreover, this rule is as powerful as the original (refine-ext) because it allows to prove a logical completeness result akin to Theorem 4.

Theorem 5 (Logical completeness of (refine-int)**).** *Consider the proof system of Fig. 3 with the addition of rules* (refine-int) *and* (limit)*. If $Q \leq [\![\mathsf{r}]\!]P$ and $[\![\mathsf{r}]\!]^A\alpha(P) = \alpha(Q)$ then $\vdash_A [P]\ \mathsf{r}\ [Q]$.*

Just like logical completeness for (refine-ext), this result implies the existence of a refinement A' in which it is possible to carry out the proof (possibly the concrete domain C). The discussion about how to find one is sketched in Section 4.3.

The second derived rule we propose is simpler than (refine-ext), as it just checks the abstractions $A(P)$ and $A'(P)$, with no reference to the regular command r nor to the postcondition Q. Since the premise is only on the precondition P, we call this rule (refine-pre).

Proposition 2. *The following rule* (refine-pre) *is extensionally sound:*

$$\frac{\vdash_{A'} [P]\ \mathsf{r}\ [Q] \quad A' \preceq A \quad A'(P) = A(P)}{\vdash_A [P]\ \mathsf{r}\ [Q]}\ (\mathsf{refine\text{-}pre})$$

Rule (refine-pre) only requires a simple check at the application site instead of an expensive analysis of the program r, so it can be preferred in practice.

We present an example to highlight the advantages of this rule (as well as (refine-int)), which allows us to use different domains in the proof derivation of different parts of the program.

Example 5 (The use of (refine-pre)*).* Consider the two program fragments

$$\begin{aligned}\mathsf{r}_1 &\triangleq \texttt{(y != 0)?; y := abs(y)}\\ \mathsf{r}_2 &\triangleq \texttt{x := y; while (x > 1) \{ y := y - 1; x := x - 1 \}}\\ &= \texttt{x := y; ((x > 1)?; y := y - 1; x := x - 1)}^*\texttt{; (x <= 1)?}\end{aligned}$$

and the program $\mathsf{r} \triangleq \mathsf{r}_1; \mathsf{r}_2$. Here `abs` is a function to compute the absolute value, and we assume, for the sake of simplicity, that the analyser knows its best abstraction. Consider the concrete domain $\mathcal{P}(\mathbb{Z}^2)$ where a pair (n, m) denote a state $\texttt{x} = n$, $\texttt{y} = m$, and the initial state $P = (\texttt{y} \in [-100; 100])$, a logical description of the concrete $\{(n, m) \mid m \in [-100; 100]\} \in \mathcal{P}(\mathbb{Z}^2)$. The bca $[\![\mathsf{r}]\!]^{\mathsf{Int}}$ in the abstract domain of intervals is locally complete on P (since P is expressible in Int), but the compositional abstraction $[\![\mathsf{r}]\!]^\sharp_{\mathsf{Int}}$ is not:

$$\begin{aligned}[\![\mathsf{r}]\!]^{\mathsf{Int}}\alpha(P) &= \mathsf{Int}([\![\mathsf{r}_2]\!][\![\mathsf{r}_1]\!](\{(n, m) \mid m \in [-100; 100]\}))\\ &= \mathsf{Int}([\![\mathsf{r}_2]\!](\{(n, m) \mid m \in [1; 100]\}))\\ &= \mathsf{Int}(\{(1, 1)\})\\ &= ([1; 1] \times [1; 1]),\end{aligned}$$

$$\dfrac{\dfrac{\mathbb{C}^{\mathsf{Int}_{\neq 0}}_{P}(\llbracket \mathtt{y\ !=\ 0?} \rrbracket)}{\vdash_{\mathsf{Int}_{\neq 0}} [P]\ \mathtt{y\ !=\ 0?}\ [R_1]}\ (\mathsf{transfer}) \qquad \dfrac{\dfrac{\mathbb{C}^{\mathsf{Int}_{\neq 0}}_{R_1}(\llbracket \mathtt{y\ :=\ abs(y)} \rrbracket)}{\vdash_{\mathsf{Int}_{\neq 0}} [R_1]\ \mathtt{y\ :=\ abs(y)}\ [\mathtt{y} \in [1;100]]}\ (\mathsf{transfer})}{\vdash_{\mathsf{Int}_{\neq 0}} [R_1]\ \mathtt{y\ :=\ abs(y)}\ [R]}\ (\mathsf{relax})}{\vdash_{\mathsf{Int}_{\neq 0}} [P]\ \mathsf{r}_1\ [R]}\ (\mathsf{seq})$$

Fig. 5: Derivation of $\vdash_{\mathsf{Int}_{\neq 0}} [P]\ \mathsf{r}_1\ [R]$ for Example 5.

while

$$\begin{aligned}
\llbracket \mathsf{r} \rrbracket^{\sharp}_{\mathsf{Int}} \alpha(P) &= \llbracket \mathsf{r}_2 \rrbracket^{\sharp}_{\mathsf{Int}} \llbracket \mathsf{r}_1 \rrbracket^{\sharp}_{\mathsf{Int}} ([-\infty;+\infty] \times [-100;100]) \\
&= \llbracket \mathsf{r}_2 \rrbracket^{\sharp}_{\mathsf{Int}} \llbracket \mathtt{y\ :=\ abs(y)} \rrbracket^{\mathsf{Int}} ([-\infty;+\infty] \times [-100;100]) \\
&= \llbracket \mathsf{r}_2 \rrbracket^{\sharp}_{\mathsf{Int}} ([-\infty;+\infty] \times [0;100]) \\
&= ([1;1] \times [0;100]) \neq ([1;1] \times [1;1]).
\end{aligned}$$

The issues are twofold. First, the analysis of r_1 in Int is incomplete, so we need a more concrete domain. For instance $\mathsf{Int}_{\neq 0}$, the Moore closure of Int with the addition of the element $\mathbb{Z}_{\neq 0}$ representing the property of being nonzero would work. Intuitively, $\mathsf{Int}_{\neq 0}$ contains all intervals, possibly having a "hole" in 0. Formally

$$\mathsf{Int}_{\neq 0} = \mathsf{Int} \cup \{ I_{\neq 0} \mid I \in \mathsf{Int} \}$$

with $\gamma'(I_{\neq 0}) = \gamma(I) \setminus \{0\}$. However, note that there is no need for a relational domain to analyze r_1 since variable $\mathtt{x}$ is never mentioned in it. On the contrary, the analysis of r_2 requires a relational domain to track the information that the value of variable $\mathtt{x}$ is equal to the value of variable $\mathtt{y}$. This suggests, for instance, to use the octagons domain Oct [15] to analyze r_2. It is worth noting that the domain of octagons Oct would not be able to perform a locally complete analysis of r_1 for the same reasons that the domain Int could not.

However, rule (refine-pre) allows us to combine these different proof derivations. Since the program state between r_1 and r_2 can be precisely represented in Int, we use this domain as a baseline and refine it in $\mathsf{Int}_{\neq 0}$ and Oct for the two parts respectively.

Let $R = (\mathtt{y} \in \{1;2;100\})$ that is an under-approximation of the concrete state in between r_1 and r_2 with the same abstraction in Int, so we can prove the triple $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}_1\ [R]$. Note that the concrete point 2 was added to R in order to have local completeness for `(x > 1)?` in r_2. However, this triple cannot be proved in Int because $\llbracket \mathsf{r}_1 \rrbracket^{\sharp}_{\mathsf{Int}}$ is not locally complete on P, so we resort to (refine-pre) to change the domain to $\mathsf{Int}_{\neq 0}$. The full derivation in $\mathsf{Int}_{\neq 0}$ is shown in Fig. 5, where $R_1 = (\mathtt{y} \in [-100;100] \wedge \mathtt{y} \neq 0)$ and we omitted for simplicity the additional hypothesis of (relax).

Again $\llbracket \mathsf{r}_2 \rrbracket$ is locally complete on R in Int, but the compositional analysis $\llbracket \mathsf{r}_2 \rrbracket^{\sharp}_{\mathsf{Int}}$ is not. Hence to perform the derivation we resort to (refine-pre) to introduce relational information in the abstract domain, using Oct instead of Int. Let

$Q = (\mathtt{x} = 1 \land \mathtt{y} = 1)$, that is the concrete output of the program, so that we can prove $\vdash_{\mathsf{Int}} [R]\ \mathsf{r}_2\ [Q]$. The derivation of this triple is only in Appendix A, Fig. 6. However, the proof is just a straightforward application of rules (seq), (iterate) and (transfer).

With those two derivation, the proof of the triple $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [Q]$ is straightforward using (refine-pre):

$$\dfrac{\dfrac{\vdash_{\mathsf{Int}_{\neq 0}} [P]\ \mathsf{r}_1\ [R]}{\vdash_{\mathsf{Int}} [P]\ \mathsf{r}_1\ [R]}\ (\mathsf{refine\text{-}pre}) \quad \dfrac{\vdash_{\mathsf{Oct}} [R]\ \mathsf{r}_2\ [Q]}{\vdash_{\mathsf{Int}} [R]\ \mathsf{r}_2\ [Q]}\ (\mathsf{refine\text{-}pre})}{\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [Q]}\ (\mathsf{seq})$$

For the derivation to fit the page, we write here the additional hypotheses of the rules. For the first application, $\mathsf{Int}_{\neq 0} \preceq \mathsf{Int}$ and $\mathsf{Int}_{\neq 0}(P) = P = \mathsf{Int}(P)$. For the second, $\mathsf{Oct} \preceq \mathsf{Int}$ and $\mathsf{Int}(R) = (\mathtt{y} \in [1; 100]) = \mathsf{Oct}(R)$.

It is worth noting that, in this example, all applications of (refine-pre) can be replaced by (refine-int). This means that also the latter is able to exploit $\mathsf{Int}_{\neq 0}$ and Oct to prove the triple in the very same way, but its application requires more expensive abstract analyses than the simple checks of (refine-pre). $\square$

While (refine-pre) is simpler than (refine-ext) and (refine-int), it is also weaker in both a theoretical and practical sense. On the one hand, LCL_A extended with this rule does not admit a logical completeness result; on the other hand, there are situations in which, even though (refine-pre) allows a derivation, the other rules are more effective. We show these two points by examples. For the first, we propose a sound triple that LCL_A extended with (refine-pre) cannot prove. Since the example is quite technical, here we only sketch the idea, and leave the details only in Appendix A, Example 8.

Example 6 (Logical incompleteness of (refine-pre)*).* Consider the concrete domain $C = \mathcal{P}(\mathbb{Z})$ of integers, the abstract domain Int of intervals, the concrete point $P = \{-1, 1\}$ and commands $\mathsf{r}_1 \triangleq$ `x != 0?`, $\mathsf{r}_2 \triangleq$ `x >= 0?` and $\mathsf{r} \triangleq \mathsf{r}_1; \mathsf{r}_2$. Then the triple $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}_1; \mathsf{r}_2\ [\{1\}]$ is sound but cannot be proved in LCL_A extended with (refine-pre).

The key observations for this example are two. First, all strict subset $P' \subset P$ are such that $\mathsf{Int}(P') \subset \mathsf{Int}(P)$. Moreover, for all refinements $A' \preceq \mathsf{Int}$ such that $A'(P) = \mathsf{Int}(P)$ we have the same condition, namely if $P' \subset P$ then $A'(P') \subset A'(P)$. This is because for all $P' \subset P$ we have $A'(P') \subseteq \mathsf{Int}(P') \subset \mathsf{Int}(P) = A'(P)$. Second, $[\![\mathsf{r}_1]\!]P = P$. This means that all triples appearing in the derivation tree of $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}_1; \mathsf{r}_2\ [\{1\}]$ have the same precondition P. Since (refine-pre) requires $A'(P) = \mathsf{Int}(P)$, all possible applications of this rule change the abstract domain to some A' satisfying the condition above. Since LCL_A computes under-approximations with the same abstraction of the strongest postcondition, these two observations make it impossible to under-approximate P further, both with (relax) and (refine-pre). This in turn make the triple not provable because $[\![\mathsf{r}_2]\!]$ is not locally complete on P in Int or in any refinement satisfying

$A'(P) = \mathsf{Int}(P)$:

$$\begin{aligned} A'[\![\mathsf{r}_2]\!](P) &= A'(\{1\}) \subseteq \mathsf{Int}(\{1\}) = \{1\} \\ A'[\![\mathsf{r}_2]\!]A'(P) &\supseteq [\![\mathsf{r}_2]\!]A'(P) = [\![\mathsf{r}_2]\!](\mathsf{Int}(P)) = \{0, 1\}. \end{aligned}$$

Example 8 in Appendix A exhibits the formal argument showing that this triple cannot be proved. □

As a corollary, this example (and more in general logical incompleteness) shows that is not always possible to find a refinement A' to carry out the proof using (refine-pre). Another consequence of this incompleteness result is the fact that, even when a command is locally complete in an abstract domain A, we may need to reason about properties that are not expressible in A in order to prove it, as (refine-pre) may not be sufficient.

Second, we present an example to illustrate that there are situations in which (refine-int) is more practical than (refine-pre), even though they are both able to prove the same triple.

Example 7. Consider the two program fragments

$\mathsf{r}_1 \triangleq$ `(y != 0)?; x := y; y := abs(y)`

$\mathsf{r}_2 \triangleq$ `x := y; while (x > 1) { y := y - 1; x := x - 1 }`

and the program $\mathsf{r} \triangleq \mathsf{r}_1; \mathsf{r}_2$. Consider also the initial state $P = \mathtt{y} \in [-100; 100]$.

This example is a variation of Example 5: the difference is the introduction of the relational dependency `x := y` in r_1, that is partially stored in the post-condition R of r_1. Because of this, $\mathsf{Oct}(R)$ and $\mathsf{Int}(R)$ are different, so we cannot apply (refine-pre) to prove $[R]\ \mathsf{r}_2\ [Q]$ for some Q.

Following Example 5, the domain $\mathsf{Int}_{\neq 0}$ is able to infer on r_1 a subset R of the strongest postcondition $\mathtt{y} \in [1; 100] \wedge \mathtt{y} = \mathrm{abs}(\mathtt{x})$ with the same abstraction $\mathsf{Int}_{\neq 0}(R) = [-100; 100]_{\neq 0} \times [1; 100]$. However, for any such R we cannot use (refine-pre) to prove the triple $\vdash_{\mathsf{Int}} [R]\ \mathsf{r}_2\ [\mathtt{x} = 1 \wedge \mathtt{y} = 1]$ via Oct because $\mathsf{Int}(R) = \mathtt{x} \in [-100; 100] \wedge \mathtt{y} \in [1; 100]$ while $\mathsf{Oct}(R) = 1 \leq \mathtt{y} \leq 100 \wedge -\mathtt{y} \leq \mathtt{x} \leq \mathtt{y}$. More in general, any subset of the strongest postcondition contains the relational information $\mathtt{y} = \mathrm{abs}(\mathtt{x})$, so relational domains like octagons and polyhedra [9] do not have the same abstraction as the non-relational Int, preventing the use of (refine-pre). However, we can apply (refine-int): considering $R = (\mathtt{y} \in \{1; 2; 100\} \wedge \mathtt{y} = \mathrm{abs}(\mathtt{x}))$, $Q = (\mathtt{x} = 1 \wedge \mathtt{y} = 1)$ and $\mathsf{r}_w \triangleq$ `while (x > 1) { y := y - 1; x := x - 1 }`, we have

$$\begin{aligned} \mathsf{Int}[\![\mathsf{r}_2]\!]^\sharp_{\mathsf{Oct}}\mathsf{Int}(R) &= \mathsf{Int}[\![\mathsf{r}_2]\!]^\sharp_{\mathsf{Oct}}(\mathtt{x} \in [-100; 100] \wedge \mathtt{y} \in [1; 100]) \\ &= \mathsf{Int}[\![\mathsf{r}_w]\!]^\sharp_{\mathsf{Oct}}[\![\mathtt{x := y}]\!]^\sharp_{\mathsf{Oct}}(\mathtt{x} \in [-100; 100] \wedge \mathtt{y} \in [1; 100]) \\ &= \mathsf{Int}[\![\mathsf{r}_w]\!]^\sharp_{\mathsf{Oct}}(1 \leq \mathtt{y} \leq 100, \mathtt{y} = \mathtt{x}) \\ &= \mathsf{Int}(\mathtt{x} = 1 \wedge \mathtt{y} = 1) \\ &= \mathsf{Int}(Q). \end{aligned}$$

In this example, rule (refine-pre) can be applied to prove the triple, but it requires to have relational information from the assignment `x := y` in r_1, hence forcing the use of a relational domain (eg. the reduced product [7] of Oct and $\mathsf{Int}_{\neq 0}$) for the whole r, making the analysis more expensive. □

4.3 Choosing The Refinement

All three new rules allow to combine different domains in the same derivation, but do not define an algorithm because of the choice of the right refinement to use is nondeterministic. A crucial point to their applicability is a strategy to select the refined abstract domain. While we have not addressed this problem yet, we believe there are some interesting starting points in the literature.

As already anticipated in previous sections, we settled the question from a theoretical point of view. Logical completeness results for (refine-ext) (Theorem 4) and (refine-int) (Theorem 5) implies the existence of a domain in which it is possible to complete the proof (if this were not the case, then the proof could not be completed in any domain, against the logical completeness). However, the proofs of those theorems exhibit the concrete domain C as an example, which is unfeasible in general. Dually, as (refine-pre) is logically incomplete (Example 6), there are triples that cannot be proved in any domain with it.

As more practical alternatives, we envisage some possibilities. First, we are studying relationships with counterexample-guided abstraction refinement (CEGAR) [4], which is a technique that exploits refinement in the context of abstract model checking. However, CEGAR and our approach seem complementary. On the one hand, our refinement rules allow a dynamic change of domain, during the analysis and only for a part of it, while CEGAR performs a static refinement and then a new analysis of the whole transition system in the new, more precise domain. On the other hand, our rules lack an instantiation technique, while for CEGAR there are effective algorithms available to pick a suitable refinement.

Second, local completeness shell [3] were proposed as an analogous of completeness shell [11] for local completeness. In the article, the authors proposed to use local completeness shells to perform abstract interpretation repair, a technique to refine the abstract domain depending on the program to analyse, just like CEGAR does for abstract model checking. Abstract interpretation repair works well with LCL_A, and could be a way to decide the best refinement for one of our rules in presence of a failed local completeness proof obligation. The advantage of combining repair with our new rules is given by the possibility of discarding the refined domain just after its use in a subderivation instead of using it to carry out the whole derivation. Investigations in this direction is ongoing.

Another related approach, which shares some common ground with CEGAR, is Lazy (Predicate) Abstraction [12,14]. Both ours and this approach exploits different abstract domains for different parts of the proof, refining it as needed. The key difference is that Lazy Abstraction unwinds the control flow graph (CFG) of the program (with techniques to handle loops) while we work inductively on the syntax. This means that, when Lazy Abstraction refines a domain, it must use it from that point onward (unless it finds a loop invariant). On the other

Proof system	Extensional	Logical completeness
Plain LCL_A	✗	✗
LCL_A + (refine-ext)	✓	✓
LCL_A + (refine-int)	✓	✓
LCL_A + (refine-pre)	✓	✗

Table 1: Comparison of the proof systems

hand, our method can change abstract domain even for different parts of sequential code. However, the technique used in Lazy Abstraction (basically to trace a counterexample back with a theorem prover until it is either found to be spurious or proved to be true) could be applicable to LCL_A: a failed local completeness proof obligation in (transfer) can be traced back with a theorem prover and the failed proof can be used to understand how to refine the abstract domain.

5 Conclusions

In this paper, we have proposed a logical framework to prove both correctness and incorrectness of a program exploiting locally complete abstractions. Indeed, from any provable triple $[P]$ r $[Q]$ we can either prove that r meets an expressible specification Spec or find a concrete counterexample in Q. Differently from the original LCL_A [2], that was proved to be *intensionally* sound, our framework is *extensionally* sound, meaning that is able to prove more properties about programs. To achieve this, our inference rules are based on the best correct abstraction of a program behaviour instead of a generic abstract interpreter. The key feature of our proof systems is the ability to exploit different abstract domains to analyse different portions of the whole program. In particular, the domains are selected among the refinements of a chosen abstract domain from which the analysis begins. The main advantage of our extensional approach is the possibility of proving many triples that could not be proved in LCL_A because of the way the program is written. More in details, we presented three new rules to refine the abstract domain, each of which can be added independently to the proof system with different complexity-precision trade-off.

Table 1 summarizes the properties LCL_A enjoys when extended with different rules, and Figure 1 from the Introduction graphically compare the logical strength of these proof systems. (refine-ext) is the most general rule, from which the other two (refine-int) and (refine-pre) are derived. The former turns out to be as strong as (refine-ext), since they are both logically complete, while the latter is simpler to use, although weaker.

Future work. In principle completeness could be achieved either refining or simplifying the abstract domain [11]. In this article we have only focused on refinement rules for local completeness, but we are investigating some simplification rules as well as their relation to the ones presented in this paper. To date, domain simplification seems theoretically weaker, but apparently it can accommodate for techniques useful in practice that are beyond the reach of refinement rules.

While the new rules we introduced are relevant from both a theoretical and practical point of view, they do not define an algorithm because of their nondeterminism: we need techniques to determine *when* a change of abstract domain is needed and *how* to choose the most convenient new domain. We believe these two issues are actually related. For instance, if the analysis is unable to satisfy a local completeness proof obligation to apply (transfer), an heuristics may determine both what additional information is needed to make it true (i.e., how to refine the abstract domain) and where that additional information came from (i.e., when to refine). We briefly discussed in Section 4.3 some possibilities to perform this choice. Ideally, one would systematically select an off-the- shelf abstract domain best suited to deal with each code fragment and the heuristic would inspect the proof obligations, and exploit some sort of catalog that can track suitable abstract domains that are locally complete for the code and input at hand or derive on-the-fly some convenient domain refinement as done, e.g., by partition refinement. To this aim, we intend to investigate a mutual exchange of ideas between CEGAR and our approach, and to integrate abstract interpretation repair into our framework.

Acknowledgments. We thank the anonymous referees for their helpful comments that helped us to improve the presentation and the discussion with related work.

Appendix A Proofs and Supplementary Material

A.1 Extensional Soundness (Theorem 2)

Proof (Proof of Theorem 2). First we remark that points (1) and (3) implies point (2):

$$\begin{aligned} \alpha(Q) &\leq \alpha([\![\mathsf{r}]\!]P) && [(1) \text{ and monotonocity of } \alpha] \\ &\leq [\![\mathsf{r}]\!]^{A}\alpha(P) && [\text{soundness of } [\![\mathsf{r}]\!]^{A}] \\ &= \alpha(Q) && [(3)] \end{aligned}$$

So all the lines are equal, in particular $\alpha(Q) = \alpha([\![\mathsf{r}]\!]P)$. The proof is then by induction on the derivation tree of $\vdash_A [P]\ \mathsf{r}\ [Q]$, but we only have to prove (1) and (3) because of the observation above. We only include one inductive case as an example, others are standard.
(seq): (1) $Q \leq [\![\mathsf{r}_2]\!]R \leq [\![\mathsf{r}_2]\!]([\![\mathsf{r}_1]\!]P) = [\![\mathsf{r1}; \mathsf{r}_2]\!]P$, where the inequalities follow from inductive hypotheses and monotonicity of $[\![\mathsf{r}_2]\!]$.

(3) We recall that $[\![\mathsf{r}_1;\mathsf{r}_2]\!]^A \leq [\![\mathsf{r}_2]\!]^A[\![\mathsf{r}_1]\!]^A$.

$$\begin{aligned}
\alpha(Q) &\leq \alpha([\![\mathsf{r}_1;\mathsf{r}_2]\!]P) && [(1) \text{ and monotonicity of } \alpha]\\
&\leq [\![\mathsf{r}_1;\mathsf{r}_2]\!]^A\alpha(P) && [\text{soundness of } [\![\mathsf{r}]\!]^A]\\
&\leq [\![\mathsf{r}_2]\!]^A[\![\mathsf{r}_1]\!]^A\alpha(P) && [\text{recalled above}]\\
&= [\![\mathsf{r}_2]\!]^A\alpha(R) && [\text{inductive hp}]\\
&= \alpha(Q) && [\text{inductive hp}]
\end{aligned}$$

So all the lines are equal, in particular $[\![\mathsf{r}_1;\mathsf{r}_2]\!]^A\alpha(P) = \alpha(Q)$.

□

A.2 Soundness and Completeness of (refine-ext)

This technical lemma is used in the following proofs.

Lemma 1. *If $A' \preceq A$ then $A = AA' = A'A$*

Proof. Fix a concrete element $c \in C$. Since $A' \preceq A$ we have $c \leq A'(c) \leq A(c)$. Applying A, by monotonicity we get $A(c) \leq AA'(c) \leq AA(c) = A(c)$, where the last equality is idempotency of A. This means $A = AA'$. Now consider $A'A(c)$. Since A is a closure operator $A'A(c) \leq A(A'A(c))$. But we just showed $AA'(A(c)) = A(A(c)) = A(c)$. Lastly, since A' is a closure operator too, $A(c) \leq A'A(c)$. Hence $A(c) \leq A'A(c) \leq A(c)$, so $A(c) = A'A(c)$.

We point out that, by injectivity of γ, this also means $\alpha\gamma'\alpha' = \alpha$.

Proof (Proof of Theorem 3). We recall that the intuitive premise $A[\![\mathsf{r}]\!]^{A'}A(P) = A(Q)$ of the rule formally is $\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha'A(P) = \alpha(Q)$. Since the proof of Theorem 2 is by induction, we can extend it just proving the inductive case for (refine-ext).
(1) It's the same as point (1) of extensional soundness (Theorem 2) applied to $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$, since this conclusion does not depend on the abstract domain.
(2-3)

$$\begin{aligned}
\alpha(Q) &\leq \alpha([\![\mathsf{r}]\!]P) && [(1) \text{ and monotonicity of } \alpha]\\
&\leq [\![\mathsf{r}]\!]^A\alpha(P) && [\text{soundness of } [\![\mathsf{r}]\!]^A]\\
&= \alpha[\![\mathsf{r}]\!]\gamma\alpha(P) && [\text{definition}]\\
&= \alpha\gamma'\alpha'[\![\mathsf{r}]\!]\gamma'\alpha'\gamma\alpha(P) && [\text{Lemma } 1]\\
&= \alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha'A(P) && [\text{definition}]\\
&= \alpha(Q) && [\text{hypothesis of the rule}]
\end{aligned}$$

Hence all the lines are equal; in particular $\alpha([\![r]\!]P) = \alpha(Q)$ and $[\![r]\!]^A\alpha(P) = \alpha(Q)$. □

Proof (Proof of Theorem 4). First, the hypotheses of the theorem implies $\mathbb{C}^A_P(\llbracket \mathsf{r} \rrbracket)$:

$$\begin{aligned} \llbracket \mathsf{r} \rrbracket^A \alpha(P) &= \alpha(Q) && \text{[hp of the theorem]} \\ &\le \alpha(\llbracket \mathsf{r} \rrbracket P) && \text{[monotonicity of } \alpha \text{ and hp of the theorem } Q \le \llbracket \mathsf{r} \rrbracket P] \\ &\le \llbracket \mathsf{r} \rrbracket^A \alpha(P) && \text{[soundness of } \llbracket \mathsf{r} \rrbracket^A] \end{aligned}$$

Hence $\alpha(\llbracket \mathsf{r} \rrbracket P) = \llbracket \mathsf{r} \rrbracket^A \alpha(P) = \alpha \llbracket \mathsf{r} \rrbracket \gamma\alpha(P)$, that is local completeness. Moreover $\alpha(Q) = \alpha(\llbracket \mathsf{r} \rrbracket P)$.

Now consider a triple P, r, Q satisfying the hypotheses. If $Q < \llbracket \mathsf{r} \rrbracket P$, using (relax) we get

$$\frac{P \le P \le A(P) \quad \vdash_A [P]\ \mathsf{r}\ [\llbracket \mathsf{r} \rrbracket P] \quad Q \le \llbracket \mathsf{r} \rrbracket P \le A(Q)}{\vdash_A [P]\ \mathsf{r}\ [Q]}\ (\mathsf{relax})$$

But the first condition is trivial, and the third one is made of $Q \le \llbracket r \rrbracket P$ (the hypothesis) and $\llbracket r \rrbracket P \le A(Q)$, that follows because $\alpha(\llbracket r \rrbracket P) = \alpha(Q)$ (shown above) and in a GC this implies $\llbracket r \rrbracket P \le \gamma\alpha(Q) = A(Q)$. Hence without loss of generality we can assume $Q = \llbracket \mathsf{r} \rrbracket P$.

Now we want to apply (refine-ext) to move to the concrete domain C. Clearly $C \preceq A$. The last hypothesis of the rule can be readily verified recalling that $\llbracket r \rrbracket^C = \llbracket r \rrbracket$ and $\alpha' = \gamma' = \mathrm{id}_C$:

$$\begin{aligned} \alpha \llbracket r \rrbracket^C A(P) &= \alpha \llbracket r \rrbracket A(P) \\ &= \llbracket r \rrbracket^A \alpha(P) \\ &= \alpha(\llbracket \mathsf{r} \rrbracket P) \end{aligned}$$

so if we can show $\vdash_C [P]\ \mathsf{r}\ [\llbracket \mathsf{r} \rrbracket P]$ we can apply (refine-ext) to prove the triple $\vdash_A [P]\ \mathsf{r}\ [\llbracket \mathsf{r} \rrbracket P]$:

$$\frac{\vdash_C [P]\ \mathsf{r}\ [\llbracket \mathsf{r} \rrbracket P] \quad C \preceq A \quad A\llbracket \mathsf{r} \rrbracket^C A(P) = A(\llbracket \mathsf{r} \rrbracket P)}{\vdash_A [P]\ \mathsf{r}\ [\llbracket \mathsf{r} \rrbracket P]}\ (\mathsf{refine\text{-}ext})$$

Lastly, we resort to logical completeness of LCL_A (cf. [2], Th 5.11) to say that the triple $\vdash_C [P]\ \mathsf{r}\ [\llbracket \mathsf{r} \rrbracket P]$ is provable. The hypothesis of that theorem are satisfied: all expressions are globally complete in the concrete domain C, $\llbracket r \rrbracket P \le \llbracket r \rrbracket P$ and $\llbracket r \rrbracket^\sharp_C \mathrm{id}_C(P) = \llbracket \mathsf{r} \rrbracket P = \mathrm{id}_C(\llbracket r \rrbracket P)$, where we used $\alpha' = \mathrm{id}_C$ and $\llbracket r \rrbracket^\sharp_C = \llbracket r \rrbracket$. □

A.3 Derived Refinement Rules

Proof (Proof of Proposition 1). We show that the hypotheses of (refine-int) implies those of (refine-ext). This means than whenever we can apply the former we could also apply the latter, that in turn means Theorem 3 ensures extensional soundness.

The first two hypotheses $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$ and $A' \preceq A$ are shared among the two rules, so we only have to show $\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha' A(P) = \alpha(Q)$. We recall that $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$ implies $Q \le [\![\mathsf{r}]\!]P$ by extensional soundness.

$$\begin{aligned}
\alpha(Q) &\le \alpha([\![\mathsf{r}]\!]P) && [Q \le [\![\mathsf{r}]\!]P \text{ and monotonicity of } \alpha]\\
&\le [\![\mathsf{r}]\!]^{A}\alpha(P) && [\text{soundness of } [\![\mathsf{r}]\!]^{A}]\\
&= \alpha[\![\mathsf{r}]\!]A(P) && [\text{definition}]\\
&= \alpha\gamma'\alpha'[\![\mathsf{r}]\!]A'A(P) && [\text{Lemma 1}]\\
&= \alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha' A(P) && [\text{definition}]\\
&\le \alpha\gamma'[\![\mathsf{r}]\!]^{\sharp}_{A'}\alpha' A(P) && [[\![\mathsf{r}]\!]^{A'} \le [\![\mathsf{r}]\!]^{\sharp}_{A'}]\\
&= \alpha(Q) && [\text{Last hypothesis of the rule}]
\end{aligned}$$

Hence all the lines are equal, and in particular $\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha' A(P) = \alpha(Q)$. □

Proof (Proof of Theorem 5). The proof is the same as that of Theorem 4, the only difference being that to apply (refine-int) we need to show $A[\![\mathsf{r}]\!]^{\sharp}_{C}A(P) = A([\![\mathsf{r}]\!]P)$ instead of $A[\![\mathsf{r}]\!]^{C}A(P) = A([\![\mathsf{r}]\!]P)$. However, since in the concrete domain $[\![\mathsf{r}]\!]^{\sharp}_{C} = [\![\mathsf{r}]\!]^{C} = [\![\mathsf{r}]\!]$ the proof still holds. □

Proof (Proof of Proposition 2). As in the proof or Proposition 1 above, we show that the hypotheses of (refine-pre) implies those of (refine-ext).

The first two hypotheses $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$ and $A' \preceq A$ are shared among the two rules, so we only have to show $\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha' A(P) = \alpha(Q)$. We recall that $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$ implies by extensional soundness (1) $Q \le [\![\mathsf{r}]\!]P$ and (3) $[\![\mathsf{r}]\!]^{A'}\alpha'(P) = \alpha'(Q)$.

$$\begin{aligned}
\alpha(Q) &\le \alpha([\![\mathsf{r}]\!]P) && [Q \le [\![\mathsf{r}]\!]P \text{ and monotonicity of } \alpha]\\
&\le [\![\mathsf{r}]\!]^{A}\alpha(P) && [\text{soundness of } [\![\mathsf{r}]\!]^{A}]\\
&= \alpha[\![\mathsf{r}]\!]A(P) && [\text{definition}]\\
&= \alpha[\![\mathsf{r}]\!]A'(P) && [\text{hp of the rule}]\\
&= \alpha\gamma'\alpha'[\![\mathsf{r}]\!]A'(P) && [\text{Lemma 1}]\\
&= \alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha'(P) && [\text{definition}]\\
&= \alpha\gamma'\alpha'(Q) && [\text{extensional soundness (3)}]\\
&= \alpha(Q) && [\text{Lemma 1}]
\end{aligned}$$

Hence all the lines are equal, and in particular $\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha' A(P) = \alpha(Q)$. □

Details about Example 5. The full derivation of the triple $\vdash_{\mathsf{Oct}} [R]\ \mathsf{r}_2\ [Q]$ for Example 5 is shown in Fig. 6, rotated and split to fit the page. The command r_i = `(x > 1)?; y := y - 1; x := x - 1` is iterated with the Kleene star and we let $R_2 = (\mathtt{y} \in \{1;2;100\} \wedge \mathtt{x} = \mathtt{y})$. We also used the logical implication $R_2 \implies (\mathtt{y} \in \{1;99\} \wedge \mathtt{x} = \mathtt{y})$, both explicitly and implicitly in the equivalence $R_2 \vee (\mathtt{y} \in \{1;99\} \wedge \mathtt{x} = \mathtt{y}) = R_2$.

$$\dfrac{\dfrac{\mathbb{C}^{\mathsf{Oct}}_{\mathtt{y}\in\{2;100\}\wedge\mathtt{x}=\mathtt{y}}(\llbracket \mathtt{y\ :=\ y\ -\ 1}\rrbracket)}{\vdash_{\mathsf{Oct}} [\mathtt{y} \in \{2;100\} \wedge \mathtt{x} = \mathtt{y}]\ \mathtt{y\ :=\ y\ -\ 1}\ [\mathtt{y}\in\{1;99\}\wedge \mathtt{x}-1=\mathtt{y}]}\ (\mathsf{transfer}) \quad \dfrac{\mathbb{C}^{\mathsf{Oct}}_{\mathtt{y}\in\{1;99\}\wedge\mathtt{x}-1=\mathtt{y}}(\llbracket \mathtt{x\ :=\ x\ -\ 1}\rrbracket)}{\vdash_{\mathsf{Oct}} [\mathtt{y} \in \{1;99\} \wedge \mathtt{x}-1 = \mathtt{y}]\ \mathtt{x\ :=\ x\ -\ 1}\ [\mathtt{y}\in\{1;99\}\wedge \mathtt{x}=\mathtt{y}]}\ (\mathsf{transfer})}{(**)}\ (\mathsf{seq})$$

$$\dfrac{\dfrac{\dfrac{\mathbb{C}^{\mathsf{Oct}}_{R_2}(\llbracket \mathtt{(x\ >\ 1)?}\rrbracket)}{\vdash_{\mathsf{Oct}} [R_2]\ \mathtt{(x\ >\ 1)?}\ [\mathtt{y}\in\{2;100\}\wedge\mathtt{x}=\mathtt{y}]}\ (\mathsf{transfer}) \quad \dfrac{(**)}{\vdash_{\mathsf{Oct}} [\mathtt{y}\in\{2;100\}\wedge\mathtt{x}=\mathtt{y}]\ \mathtt{y\ :=\ y\ -\ 1;\ x\ :=\ x\ -\ 1}\ [\mathtt{y}\in\{1;99\}\wedge\mathtt{x}=\mathtt{y}]}\ (\mathsf{seq})}{\vdash_{\mathsf{Oct}} [R_2]\ \mathsf{r}_i\ [\mathtt{y}\in\{1;99\}\wedge\mathtt{x}=\mathtt{y}]}\ (\mathsf{seq}) \qquad (\mathtt{y}\in\{1;99\}\wedge\mathtt{x}=\mathtt{y}) \le A(R_2)}{(*)}\ (\mathsf{iterate})$$

$$\dfrac{\dfrac{\mathbb{C}^{\mathsf{Oct}}_{R}(\llbracket \mathtt{x\ :=\ y}\rrbracket)}{\vdash_{\mathsf{Oct}} [R]\ \mathtt{x\ :=\ y}\ [R_2]}\ (\mathsf{transfer}) \quad \dfrac{\dfrac{(*)}{\vdash_{\mathsf{Oct}} [R_2]\ \mathsf{r}_i^*\ [R_2 \vee (\mathtt{y}\in\{1;99\}\wedge\mathtt{x}=\mathtt{y})]}\ (\mathsf{iterate}) \quad \dfrac{\mathbb{C}^{\mathsf{Oct}}_{R_2}(\llbracket \mathtt{(x\ <=\ 1)?}\rrbracket)}{\vdash_{\mathsf{Oct}} [R_2]\ \mathtt{(x\ <=\ 1)?}\ [Q]}\ (\mathsf{transfer})}{\vdash_{\mathsf{Oct}} [R_2]\ \mathsf{r}_i^*\mathtt{;\ (x\ <=\ 1)?}\ [Q]}\ (\mathsf{seq})}{\vdash_{\mathsf{Oct}} [R]\ \mathsf{r}_2\ [Q]}\ (\mathsf{seq})$$

Fig. 6: Derivation of $\vdash_{\mathsf{Oct}} [R]\ \mathsf{r}_2\ [Q]$ for Example 5.

$$\dfrac{\dfrac{\dfrac{\mathbb{C}^{\mathsf{Int}_P}_P(\llbracket \texttt{x != 0?} \rrbracket)}{\vdash_{\mathsf{Int}_P} [P]\ \texttt{x != 0?}\ [P]}\ (\mathsf{transfer}) \qquad \dfrac{\mathbb{C}^{\mathsf{Int}_P}_P(\llbracket \texttt{x >= 0?} \rrbracket)}{\vdash_{\mathsf{Int}_P} [P]\ \texttt{x >= 0?}\ [Q]}\ (\mathsf{transfer})}{\vdash_{\mathsf{Int}_P} [P]\ \mathsf{r}_1;\mathsf{r}_2\ [Q] \quad \mathsf{Int}_P \preceq \mathsf{Int} \quad \mathsf{Int}(\llbracket\mathsf{r}\rrbracket^\sharp_{\mathsf{Int}_P}(\mathsf{Int}(P))) = \mathsf{Int}(Q)}\ (\mathsf{seq})}{\vdash_{\mathsf{Int}} [P]\ \mathsf{r}_1;\mathsf{r}_2\ [Q]}\ (\mathsf{refine\text{-}int})$$

Fig. 7: Derivation of $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [Q]$ for Example 8.

Example 8 (Supplement to Example 6). Consider the concrete domain $C = \mathcal{P}(\mathbb{Z})$ of integers, the abstract domain Int of intervals, the concrete points $P = \{-1, 1\}$ and $Q = \{1\}$, commands $\mathsf{r}_1 \triangleq \texttt{x != 0?}$, $\mathsf{r}_2 \triangleq \texttt{x >= 0?}$ and $\mathsf{r} \triangleq \mathsf{r}_1; \mathsf{r}_2$. Let $f_1 = \llbracket\mathsf{r}_1\rrbracket$, $f_2 = \llbracket\mathsf{r}_2\rrbracket$ and $f = \llbracket\mathsf{r}\rrbracket = f_2 \circ f_1$. Observe that in the concrete semantics $f_1(P) = P$ and $f(P) = f_2(P) = \{1\}$. Consider LCL$_A$ extended with ($\mathsf{refine\text{-}pre}$), and let us show that we cannot prove $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [Q]$. Inspecting the logic, we can only apply three rules to prove this triple: (relax), ($\mathsf{refine\text{-}pre}$) or ($\mathsf{seq}$). To apply rule ($\mathsf{relax}$) we would need either an under-approximation P' of P with the same abstraction, that does not exist, or an over-approximation of Q, that would be unsound since $Q = f(P)$. Hence we cannot apply (relax). Suppose to apply ($\mathsf{refine\text{-}pre}$): any A' used in the rule should satisfy $A' \preceq \mathsf{Int}$ and $A'(P) = \mathsf{Int}(P)$; as we remarked in Example 6 this means that $P' \subset P$ implies $A'(P') \subset A'(P)$. Again this means we cannot apply (relax) even after the domain refinement. The only rule that can be applied is then (seq): to do that, we must prove two triples $\vdash_{A'} [P]\ \mathsf{r}_1\ [R]$ and $\vdash_{A'} [R]\ \mathsf{r}_2\ [Q]$. Irrespective of how we prove the first triple, by soundness (Theorem 2) we have $R \subseteq f_1(P) = P$ and $A'(R) = A'(f_1(P)) = A'(P)$, so again $R = P$. Now we should prove a triple $\vdash_{A'} [P]\ \mathsf{r}_2\ [Q]$, but this is impossible since by soundness this would imply local completeness of $\llbracket\mathsf{r}_2\rrbracket = f_2$ on P in A', that does not hold:

$$\begin{aligned} A' f_2(P) &= A'(\{1\}) \subseteq \mathsf{Int}(\{1\}) = \{1\} \\ A' f_2 A'(P) &\supseteq f_2 A'(P) = f_2(\mathsf{Int}(P)) = \{0, 1\} \end{aligned}$$

Observe that, if we add ($\mathsf{refine\text{-}int}$) to the proof system, we can use it to change the domain to one where we can express P (for instance, the concrete domain $\mathcal{P}(\mathbb{Z})$ or the refinement $\mathsf{Int} \cup \{P\}$) to prove the triple applying (seq) and then ($\mathsf{transfer}$) on both subtrees, as shown in Fig. 7. □

References

1. Bruni, R., Giacobazzi, R., Gori, R., Garcia-Contreras, I., Pavlovic, D.: Abstract extensionality: On the properties of incomplete abstract interpretations. Proc. ACM Program. Lang. **4**(POPL) (dec 2019). https://doi.org/10.1145/3371096
2. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A logic for locally complete abstract interpretations. In: Proc. of LICS'21. pp. 1–13. IEEE (2021). https://doi.org/10.1109/LICS52264.2021.9470608

3. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: Abstract interpretation repair. In: Jhala, R., Dillig, I. (eds.) Proc. of PLDI'22. pp. 426–441. ACM (2022). https://doi.org/10.1145/3519939.3523453
4. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Proc. of CAV'00. pp. 154–169. Springer (2000). https://doi.org/10.1007/10722167_15
5. Cousot, P.: Principles of Abstract Interpretation. MIT Press (2021)
6. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of POPL'77. p. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. of POPL'79. p. 269–282. ACM (1979). https://doi.org/10.1145/567752.567778
8. Cousot, P., Cousot, R.: Abstract interpretation: Past, present and future. In: Proc. of CSL-LICS'14. ACM (2014). https://doi.org/10.1145/2603088.2603165
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. of POPL'78. p. 84–96. ACM (1978). https://doi.org/10.1145/512760.512770
10. Giacobazzi, R., Logozzo, F., Ranzato, F.: Analyzing program analyses. In: Rajamani, S.K., Walker, D. (eds.) Proc. of POPL'15. pp. 261–273. ACM (2015). https://doi.org/10.1145/2676726.2676987
11. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. ACM **47**(2), 361–416 (mar 2000). https://doi.org/10.1145/333979.333989
12. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Launchbury, J., Mitchell, J.C. (eds.) Proc. of POPL'02. pp. 58–70. ACM (2002). https://doi.org/10.1145/503272.503279
13. Laviron, V., Logozzo, F.: Refining abstract interpretation-based static analyses with hints. In: Hu, Z. (ed.) Proc. of APLAS'09. LNCS, vol. 5904, pp. 343–358. Springer (2009). https://doi.org/10.1007/978-3-642-10672-9_24
14. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) Proc. of CAV'06. LNCS, vol. 4144, pp. 123–136. Springer (2006). https://doi.org/10.1007/11817963_14
15. Miné, A.: The octagon abstract domain. High. Order Symb. Comput. **19**(1), 31–100 (2006). https://doi.org/10.1007/s10990-006-8609-1
16. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL) (dec 2019). https://doi.org/10.1145/3371078
17. Raad, A., Berdine, J., Dang, H., Dreyer, D., O'Hearn, P.W., Villard, J.: Local reasoning about the presence of bugs: Incorrectness separation logic. In: Lahiri, S.K., Wang, C. (eds.) Proc. of CAV'20, Part II. LNCS, vol. 12225, pp. 225–252. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_14
18. Winskel, G.: The Formal Semantics of Programming Languages: an Introduction. MIT press (1993)

Clustered Relational Thread-Modular Abstract Interpretation with Local Traces

Michael Schwarz[1](✉), Simmo Saan[2], Helmut Seidl[1], Julian Erhard[1], and Vesal Vojdani[2]

[1] Technische Universität München, Garching, Germany
{m.schwarz, helmut.seidl, julian.erhard}@tum.de
[2] University of Tartu, Tartu, Estonia
{simmo.saan, vesal.vojdani}@ut.ee

Abstract. We construct novel thread-modular analyses that track relational information for potentially overlapping clusters of global variables – given that they are protected by common mutexes. We provide a framework to systematically increase the precision of clustered relational analyses by splitting control locations based on abstractions of *local traces*. As one instance, we obtain an analysis of dynamic thread creation and joining. Interestingly, tracking *less relational* information for globals may result in higher precision. We consider the class of 2-decomposable domains that encompasses many weakly relational domains (e.g., *Octagons*). For these domains, we prove that maximal precision is attained already for clusters of globals of sizes at most 2.

Keywords: thread-modular relational abstract interpretation, collecting local trace semantics, clusters, dynamic thread creation, concurrency

1 Introduction

Tracking relationships between program variables is indispensable for proving properties of programs or verifying the absence of certain programming errors [14, 16, 33]. Inferring relational properties is particularly challenging for multithreaded programs as all interferences by other threads that may happen in parallel, must be taken into account. In such an environment, only relational properties between globals protected by common mutexes are likely to persist throughout program execution. Generally, relations on clusters consisting of fewer variables are less brittle than those on larger clusters. Moreover, monolithic relational analyses employing, e.g., the polyhedral abstract domain are known to be notoriously expensive [36, 54]. Tracking *smaller* clusters may even be more precise than tracking larger clusters [19].

Example 1. Consider the following program. All accesses to globals g, h, and i are protected by the mutex a.

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 28–58, 2023.
https://doi.org/10.1007/978-3-031-30044-8_2

```
main:                                                t1:             t2:
 x = create(t1); y = create(t2);                      lock(a);        lock(a);
 lock(a);                                             x = h;          g = ?; h = ?;
 g = ?; h = ?; i = ?;                                 i = x;          unlock(a);
 unlock(a); r = join(y); lock(a);                     unlock(a);      return 0;
 z = ?; g = z; h = z; i = z;                          return 1;
 unlock(a); lock(a);
 // ASSERT(h==i); (1) ASSERT(g==h); (2)
 unlock(a);
```

In this program, the main thread creates two new threads, starting at t_1 and t_2, respectively. Then it locks the mutex a to set all globals non-deterministically to some value and unlocks a again. After having joined the thread t_2, it locks a again and sets all globals to the *same* unknown value and unlocks a again. Thread t_1 sets i to the value of h. Thread t_2 sets g and h to (potentially different) unknown values. Assume we are interested in equalities between globals. In order to succeed in showing assertion (1), it is necessary to detect that the main thread is unique and thus cannot read its past writes since these have been overwritten. Additionally, the analysis needs to certify that thread t_2 also is unique, has been joined before the assertion, and that its writes must also have been overwritten.

For an analysis to prove assertion (2), propagating a joint abstraction of the values of all globals protected by a does not suffice: At the unlock of a in t_1, $g{=}h$ need not hold. If this monolithic relation is propagated to the last lock of a in main, (2) cannot be shown — despite t_1 modifying neither g nor h. $\square$

Here we show, that the loss of precision indicated in the example can be remedied by replacing the monolithic abstraction of all globals protected by a mutex with suitably chosen subclusters. In the example, we propose to instead consider the subclusters $\{g, h\}$ and $\{h, i\}$ separately. As t_1 does not write any values to the cluster $\{g, h\}$, the imprecise relation $\top$ is not propagated to the main thread and assertion (2) can be shown.

To fine-tune the analysis, we rely on *weakly* relational domains. A variety of weakly relational domains have been proposed in the literature such as *Two Variables Per Inequality* [53], *Octagons* [36, 37], or simplifications thereof [33, 35]. The technical property of interest which all these domains have in common is that each abstract relation can be reconstructed from its projections onto subclusters of variables of size at most 2. We call such domains 2-*decomposable*. Beyond the numerical 2-decomposable domains, also non-numerical 2-decomposable domains can be constructed such as a domain relating string names and function pointers.

Based on 2-decomposable domains, we design thread-modular relational analyses of globals which may attain additional precision by taking *local* knowledge of threads into account. Therefore, we do not rely on a *global* trace semantics, but on a *local* trace semantics which formalizes for each thread that part of the computational past it can observe [48]. Abstract values for program points describe the set of all reaching local traces. Likewise, values recorded for *observable* actions are abstractions of all local traces ending in the corresponding action. Such observable actions are, e.g., unlock operations for mutexes. The abstract values are then refined by taking finite abstractions of local traces into account. To this end, we propose a generic framework that re-uses the components of any base analysis as black boxes. Our contributions can be summarized as follows:

– We provide new relational analyses of globals as abstractions of the local trace semantics based on overlapping variable clusters (Sections 3, 4, and 8).
– Our analysis deals with dynamically created and joined threads, whose thread *id*s may, e.g., be communicated to other threads via variables and which may synchronize via mutexes (Section 3).
– We provide a generic scheme to incorporate history-based arguments into the analysis by taking finite abstractions of local traces into account (Section 5).
– We give an analysis of dynamically created thread *id*s as an instance of our generic scheme. We apply this to exclude *self-influences* or reads from threads that cannot possibly run in parallel (Sections 6 and 7).
– We prove that some loss of precision of relational analyses can be avoided by tracking *all* subclusters of variables. For the class of 2-decomposable relational domains, we prove that tracking variable clusters of size greater than 2 can be abandoned without precision loss (Section 8).

The analyses in this paper have all been implemented, a report of a practical evaluation is included in Section 9, whereas Section 10 details related work.

2 Relational Domains

First, we define the notion of relational domain employed in the description of our analysis. Let $\mathcal{V}ars$ be a set of variables, potentially of different types. We assume all configurations and assignments to be well-typed, i.e., the type of the (abstract) value matches the one specified for a variable. For each type τ of values, we assume a complete lattice $\mathcal{V}^\sharp_\tau$ of abstract values abstracting the respective concrete values from $\mathcal{V}_\tau$. Let $\mathcal{V}^\sharp$ denote the collection of these lattices, and $\mathcal{V}ars \to_\perp \mathcal{V}^\sharp$ denote the set of all type-consistent assignments σ from variables to non-$\perp$ abstract values, extended with a dedicated least element (also denoted by $\perp$), and equipped with the induced ordering. A *relational domain* $\mathcal{R}$ then is a complete lattice which provides the following operations

$$\begin{array}{rll} [\![x \leftarrow e]\!]^\sharp_{\mathcal{R}} : \mathcal{R} \to \mathcal{R} & \text{(assignment for expression } e) & \quad \mathsf{lift} : (\mathcal{V}ars \to_\perp \mathcal{V}^\sharp) \to \mathcal{R} \\ r|_Y : \mathcal{R} \to \mathcal{R} & \text{(restriction to } Y \subseteq \mathcal{V}ars) & \quad \mathsf{unlift} : \mathcal{R} \to (\mathcal{V}ars \to_\perp \mathcal{V}^\sharp) \\ [\![?e]\!]^\sharp_{\mathcal{R}} : \mathcal{R} \to \mathcal{R} & \text{(guard for condition } e) & \end{array}$$

The operations to the left provide the abstract state transformers for the basic operation of programs (with non-deterministic assignments expressed as restrictions), while lift and unlift allow casting from abstract variable assignments to the relational domain as well as extracting single-variable information. We assume that $\mathsf{lift}\,\perp = \perp$ and $\mathsf{unlift}\,\perp = \perp$, and require that $\mathsf{unlift} \circ \mathsf{lift} \sqsupseteq \mathsf{id}$ where $\sqsupseteq$ refers to the ordering of $(\mathcal{V}ars \to_\perp \mathcal{V}^\sharp)$. Moreover, we require that the *meet* operations $\sqcap$ of $\mathcal{V}^\sharp$ and $\mathcal{R}$ safely approximate the intersection of the concretizations of the respective arguments. Restricting a relation r to a subset Y of variables amounts to *forgetting* all information about variables not in Y. Thus, we demand $r|_{\mathcal{V}ars} = r$, $r|_\emptyset = \top$, $r|_{Y_1} \sqsupseteq r|_{Y_2}$ when $Y_1 \subseteq Y_2$, $(r|_{Y_1})\big|_{Y_2} = r|_{Y_1 \cap Y_2}$, and

$$\mathsf{unlift}\,(r|_Y)\,x = \top \quad (x \notin Y) \qquad\qquad \mathsf{unlift}\,(r|_Y)\,x = (\mathsf{unlift}\,r)\,x \quad (x \in Y) \quad (1)$$

Restriction thus is *idempotent*. For convenience, we also define a shorthand for assignment of abstract values[3]: $[\![x \leftarrow^\sharp v]\!]^\sharp_{\mathcal{R}}\, r = \left(r|_{\mathcal{V}ars\setminus\{x\}}\right) \sqcap (\mathsf{lift}\,(\top \oplus \{x \mapsto v\}))$. In order to construct an abstract interpretation, we further require monotonic concretization functions $\gamma_{\mathcal{V}^\sharp} : \mathcal{V}^\sharp \to 2^{\mathcal{V}}$ and $\gamma_{\mathcal{R}} : \mathcal{R} \to 2^{\mathcal{V}ars \to \mathcal{V}}$ satisfying the requirements presented in Fig. 1.

Example 2. As a value domain $\mathcal{V}^\sharp_\tau$, consider the flat lattice over the sets of values of appropriate type τ. A relational domain $\mathcal{R}_1$ is obtained by collecting satisfiable conjunctions of equalities between variables or variables and constants where the ordering is logical implication, extended with False as least element. The greatest element in this complete lattice is given by True. The operations lift and unlift for non-$\bot$ arguments then can be defined as

$$\mathsf{lift}\,\sigma = \bigwedge\{x = \sigma\,x \mid x \in \mathcal{V}ars, \sigma\,x \neq \top\} \qquad \mathsf{unlift}\,r\,x = \begin{cases} c & \text{if } r \implies (x = c) \\ \top & \text{otherwise} \end{cases}$$

The restriction of r to a subset Y of variables is given by the conjunction of all equalities implied by r which only contain variables from Y or constants. □

In line of Example 2, also non-numerical relational domains may be constructed.

A variable clustering $\mathcal{S} \subseteq 2^{\mathcal{V}ars}$ is a set of subsets (*clusters*) of variables. For any cluster $Y \subseteq \mathcal{V}ars$, let $\mathcal{R}^Y = \{r \mid r \in \mathcal{R}, r|_Y = r\}$; this set collects all abstract values from $\mathcal{R}$ containing information on variables in Y only. Given an arbitrary clustering $\mathcal{S} \subseteq 2^{\mathcal{V}ars}$, *any* relation $r \in \mathcal{R}$ can be approximated by a meet of relations from $\mathcal{R}^Y$ ($Y \in \mathcal{S}$) since for every $r \in \mathcal{R}$, $r \sqsubseteq \bigsqcap\{r|_Y \mid Y \in \mathcal{S}\}$ holds.

Some relational domains, however, can be fully recovered from their restrictions to specific subsets of clusters. We consider for $k \geq 1$, the set $\mathcal{S}_k$ of all non-empty subsets $Y \subseteq \mathcal{V}ars$ of cardinality at most k. We call a relational domain $\mathcal{R}$ *k-decomposable* if each abstract value from $\mathcal{R}$ can be precisely expressed

[3] We use $\sigma \oplus \{x_i \mapsto v_i \mid i = 1, \ldots, m\}$ to denote the variable assignment obtained from σ by replacing the values for x_i with v_i ($i = 1, \ldots, m$).

$$\forall a, b : a \sqsubseteq b \implies \gamma_{\mathcal{V}^\sharp}\,a \subseteq \gamma_{\mathcal{V}^\sharp}\,b \qquad \gamma_{\mathcal{R}}\,\bot = \emptyset \qquad \forall r, s : r \sqsubseteq s \implies \gamma_{\mathcal{R}}\,r \subseteq \gamma_{\mathcal{R}}\,s$$
$$\gamma_{\mathcal{R}}\,([\![x \leftarrow e]\!]^\sharp_{\mathcal{R}}\,r) \supseteq \{\sigma \oplus \{x \mapsto [\![e]\!]\sigma\} \mid \sigma \in \gamma_{\mathcal{R}} r\}$$
$$\gamma_{\mathcal{R}}(r|_Y) \supseteq \{\sigma \oplus \{x_1 \mapsto v_1, \ldots, x_m \mapsto v_m\} \mid v_i \in \mathcal{V}, x_i \in \mathcal{V}ars \setminus Y, \sigma \in \gamma_{\mathcal{R}} r\}$$
$$\gamma_{\mathcal{R}}\,(\mathsf{lift}\,\sigma^\sharp) \supseteq \{\sigma \mid \forall x :\ \sigma\,x \in \gamma_{\mathcal{V}^\sharp}\,(\sigma^\sharp\,x)\} \qquad \gamma_{\mathcal{V}^\sharp}\,(\mathsf{unlift}\,r)\,x \supseteq \{\sigma\,x \mid \sigma \in \gamma_{\mathcal{R}}\,r\}$$

Fig. 1: Required properties for $\gamma_{\mathcal{V}^\sharp} : \mathcal{V}^\sharp \to 2^{\mathcal{V}}$ and $\gamma_{\mathcal{R}} : \mathcal{R} \to 2^{\mathcal{V}ars \to \mathcal{V}}$.

as the meet of its restrictions to clusters of $\mathcal{S}_k$ and when all least upper bounds can be recovered by computing with clusters of $\mathcal{S}_k$ only; that is,

$$r = \bigsqcap \left\{ r|_Q \mid Q \in \mathcal{S}_k \right\} \qquad \left(\bigsqcup R\right)|_Q = \bigsqcup \left\{ r|_Q \mid r \in R \right\} \quad (Q \in \mathcal{S}_k) \qquad (2)$$

holds for each abstract relation $r \in \mathcal{R}$ and each set of abstract relations $R \subseteq \mathcal{R}$.

Example 3. The domain $\mathcal{R}_1$ from the previous example is 2-decomposable. This also holds for the *octagon* domain [36] and many other weakly relational numeric domains (pentagons [33], weighted hexagons [21], logahedra [28], TVPI [53], dDBM [46], and AVO [11]). The *affine equalities* or *affine inequalities* domains [16, 30], however, are not. The relational string domains proposed by Arceri et al. [6, Sec. 5.1 - 5.3], are examples of non-numeric 2-decomposable domains.

3 A Local Trace Semantics

We build upon the semantic framework for *local traces*, introduced by Schwarz et al. [48]. A local trace records all past events that have affected the present configuration of a specific thread, referred to as the *ego* thread. In [48], the local trace semantics is proven equivalent to the global trace semantics which itself is equivalent to a global interleaving semantics. In particular, any analysis that is sound w.r.t. the local trace semantics also is w.r.t. the interleaving semantics.

While the framework of Schwarz et al. [48] allows for different formalizations of traces, thread synchronization happens only via locking/unlocking and thread creation. Generalizing their semantics, we identify certain actions as *observable* by other threads when executing corresponding *observing* actions (see Table 1). When the *ego* thread executes an *observing* action, a local trace ending in the corresponding *observable* action is incorporated. Here, we consider as observable/observing actions locking/unlocking mutexes and creating/joining threads.

Consider, e.g., the program in Fig. 2a and a corresponding local trace (Fig. 2b). This trace consists of one *swim lane* for each thread representing the sequence of steps it executed where each node in the graph represents a configuration attained by it. Additionally, the trace records the *create* and *join* orders as well as for each mutex a, the *locking* order for a ($\rightarrow_c, \rightarrow_j$, and $\rightarrow_a$, respectively). These

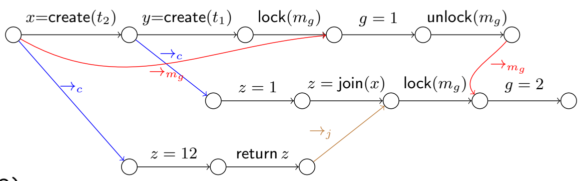
Table 1: Observable and observing actions and which concurrency primitive they relate to. The primitives targeted by this paper are in bold font.

Observable Action	Observing Action	Programming Concept
unlock(a)	lock(a)	**Mutex**, Monitor, ...
return x	x'=join(x'')	**Thread Returning / Joining**
$g = x$	$x = g$	**Writing/Reading a global variable**
`signal(c)`	`wait(c)`	Condition Variables
`send(chan,v)`	`x = receive(chan)`	Channel-Based Concurrency, Sockets, ...
`set_value`	`get`	Futures / Promises

```
main :                t1 :
 x = create(t2);      z = 1;
 y = create(t1);      z = join(x);
 lock(mg);            lock(mg);
 g = 1;               g = 2;
 unlock(mg);          unlock(mg);
 z = 28;              x = create(t2);
```

```
t2 :
 z = 12;
 return z;
```

(a) Source code



(b) Local Trace; For this program, execution begins at program point main, and x, y, z are local variables, whereas g is a global variable. To ensure atomicity, every access to the global g is protected by the mutex m_g, which we omit in the further examples.

Fig. 2: An example program and a corresponding local trace.

orders introduce extra relationships between thread configurations. The unique start node of each local trace is an initial configuration of the *main* thread.

We distinguish between the sets $\mathcal{X}$ and $\mathcal{G}$ of *local* and *global* variables. We assume that $\mathcal{X}$ contains a special variable self within which the thread *id* of the current thread, drawn from the set $\mathcal{I}$, is maintained. A (local) *thread configuration* is a pair (u, σ) where u is a program point and the type-consistent map $\sigma : \mathcal{X} \rightarrow \mathcal{V}$ provides values for the local variables. The values of globals are *not* explicitly represented in a thread configuration, but can be recovered by consulting the (unique) last write to this global within the local trace. To model weak memory effects, weaker notions of last writes are conceivable. As in [48], we consider a set of actions $\mathcal{A}ct$ that consists of locking and unlocking a (non-reentrant) mutex from a set M, copying values of globals into locals and vice-versa, creating a new thread, as well as assignments with and branching on local variables. We extend $\mathcal{A}ct$ with actions for *returning* from and *joining* with threads. We assume that writes to and reads from globals are *atomic* (or more precisely, we assume copying values of *integral* type to be atomic). This is enforced for each global g by a dedicated mutex m_g acquired just before accessing g and released immediately after. For simplicity, we associate traces corresponding to a write of g to this dedicated mutex m_g, and thus do not need to consider writing and reading of globals as observable/observing actions. In examples, we omit explicitly locking and unlocking these mutexes. By convention, at program start all globals have value 0, while local variables may initially have any value.

Each thread is represented by a control-flow graph with edges $e \in \mathcal{E}$ of the form $e = (u, \mathsf{act}, u')$ for some action $\mathsf{act} \in \mathcal{A}ct$ and program points u and u' where the start point of the *main* thread is u_0. Let $\mathcal{T}$ denote the set of all local traces of a given program. A formalism for local traces must, for each edge e of the control-flow graph, provide a transformation $[\![e]\!] : \mathcal{T}^k \rightarrow 2^{\mathcal{T}}$ so that $[\![e]\!](t_0, \ldots, t_{k-1})$ extends the local trace t_0, possibly incorporating other local traces. For the operations $\mathsf{lock}(a), a \in \mathsf{M}$, or x=join$(x'), x, x' \in \mathcal{X}$, the arity of $[\![e]\!]$ is two, another local trace, namely, with last operation $\mathsf{unlock}(a)$ or $\mathsf{return}\, x''$, respectively, is incorporated. The remaining edge transformations have arity one. In all cases, the set of resulting local traces may be empty when the operation is not applicable to its argument(s). We write $[\![e]\!](T_0, \ldots, T_{k-1})$ for the set $\bigcup_{t_0 \in T_0, \ldots, t_{k-1} \in T_{k-1}} [\![e]\!](t_0, \ldots, t_{k-1})$.

Given definitions of $[\![e]\!]$, the set $\mathcal{T}$ can be inductively defined starting from a set init of *initial local traces* consisting of initial configurations of the *main* thread. To develop efficient thread-modular abstractions, we are interested in subsets $\mathcal{T}[u], \mathcal{T}[a], \mathcal{T}[i]$ of local traces ending at some program point u, ending with an *unlock* operation for mutexes a (or from init), or ending with a *return* statement of thread i, respectively. Schwarz et al. [48] showed that such subsets can be described as the least solution of a *side-effecting constraint system* [5]. There, each right-hand side may, besides its contribution to the unknown on the left, also provide contributions to other unknowns (the *side-effects*). This allows expressing analyses that accumulate flow-insensitive information about globals during a flow-sensitive analysis of local states with dynamic control flow [51]. Here, in the presence of dynamic thread creation, we use side-effects to express that an observable action, unlock or return, should *also* contribute to the sets $\mathcal{T}[a]$ or $\mathcal{T}[i]$, such that they can be incorporated at the corresponding observing action. The side-effecting formulation of our concrete semantics takes the form:

$$(\eta, \eta\,[u_0]) \sqsupseteq (\{[a] \mapsto \mathsf{init} \mid a \in \mathsf{M}\}, \mathsf{init}) \qquad (\eta, \eta\,[u']) \sqsupseteq [\![u, \mathsf{act}]\!]\eta \quad (u, \mathsf{act}, u') \in \mathcal{E} \tag{3}$$

where the ordering $\sqsupseteq$ is induced by the superset ordering and right-hand sides are defined in Fig. 3. A right-hand side takes an assignment η of the unknowns of the system and returns a pair (η', T) where T is the contribution to the unknown occurring on the left (as in ordinary constraint systems). The first component collects the side-effects as the assignment η'. If the right-hand sides are monotonic, Eq. (3) has a unique least solution.

We only detail the right-hand sides for the *creation* of threads as well as the new actions *join* and *return*; the rest remain the same as defined by Schwarz et al. [48]. For **thread creation**, they provide the action $x{=}\mathsf{create}(u_1)$. Here, u_1 is the program point at which the created thread should start. We assume that all locals from the creator are passed to the created thread, except for the

$$
\begin{aligned}
&[\![u, \mathsf{lock}(a)]\!]\,\eta = (\emptyset, [\![e]\!](\eta\,[u], \eta\,[a])) \\
&[\![u, \mathsf{unlock}(a)]\!]\,\eta = \\
&\quad \mathbf{let}\ T = [\![e]\!](\eta\,[u])\ \mathbf{in} \\
&\quad (\{[a] \mapsto T\}, T) \\
&[\![u, x = g]\!]\,\eta = (\emptyset, [\![e]\!](\eta\,[u])) \\
&[\![u, g = x]\!]\,\eta = (\emptyset, [\![e]\!](\eta\,[u]))
\end{aligned}
\qquad
\begin{aligned}
&[\![u, x{=}\mathsf{create}(u_1)]\!]\,\eta = \mathbf{let}\ T = [\![e]\!](\eta\,[u])\ \mathbf{in} \\
&\quad (\{[u_1] \mapsto \mathsf{new}\, u\, u_1\, (\eta\,[u])\}, T) \\
&[\![u, x{=}\mathsf{join}(x')]\!]\ \eta = \mathbf{let}\ T = \eta\,[u]\ \mathbf{in} \\
&\quad (\emptyset, [\![e]\!](\eta\,[u], \textstyle\bigcup\{\eta\,[t\,(x')] \mid t \in \eta\,[u]\})) \\
&[\![u, \mathsf{return}\, x]\!]\ \eta = \mathbf{let}\ T = \eta\,[u]\ \mathbf{in} \\
&\quad (\{[i] \mapsto [\![e]\!](\{t \in T \mid t(\mathsf{self}) = i\}) \mid i \in \mathcal{I}\}, [\![e]\!]T)
\end{aligned}
$$

Fig. 3: Right-hand sides for side-effecting formulation of concrete semantics; $t(y)$ extracts the value of local variable y from the terminal configuration of trace t.

variable self. The variables self in the created thread and x in the creating thread receive a fresh thread *id*. Here, $\mathsf{new}\, u\, u_1\, t$ computes the local trace at the start point u_1 from the local trace t of the creating thread. To handle **returning** and **joining** of threads we introduce the following two actions:

– $\mathsf{return}\ x$; – terminating a thread and returning the value of the local variable x to a thread waiting for the given thread to terminate.
– x=$\mathsf{join}(x')$; where x' is a local variable holding a thread *id* – blocks the ego thread, until the thread with the given thread *id* has terminated. As in *pthreads*, at most one thread may call join for a given thread *id*. The value provided to return by the joined thread is assigned to the local variable x.

For returning results and realization of join, we employ the unknown $[i]$ for the thread *id* i of the returning thread, as shown in Fig. 3.

4 Relational Analyses as Abstractions of Local Traces

Subsequently, we give relational analyses of the values of globals which we base on the local trace semantics. They are generic in the relational domain $\mathcal{R}$, with 2-decomposable domains being particularly well-suited, as the concept of *clusters* is central to the analyses. We focus on relations between globals that are jointly *write*-protected by some mutex. We assume we are given for each global g, a set $\mathcal{M}[g]$ of (write) *protecting* mutexes, i.e., mutexes that are *always* held when g is written. Let $\mathcal{G}[a] = \{g \in \mathcal{G} \mid a \in \mathcal{M}[g]\}$ denote the set of globals protected by a mutex a. Let $\emptyset \neq \mathcal{Q}_a \subseteq 2^{\mathcal{G}[a]}$ the set of clusters of these globals we associate with a. For technical reasons, we require at least one cluster per mutex a, which may be the empty cluster $\emptyset$, thus not associating any information with a.

Our basic idea is to store at the unknown $[a, Q]$ (for each mutex a and cluster $Q \in \mathcal{Q}_a$) an abstraction of the relations *only* between globals in Q. By construction, all globals in Q are protected by a. Whenever it is locked, the relational information stored at all $[a, Q]$ is incorporated into the local state by the lattice operation *meet*, i.e., the local state now maintains relations between locals *as well as* globals which no other thread can access at this program point. Whenever a is unlocked, the new relation between globals in all corresponding clusters $Q \in \mathcal{Q}_a$ is side-effected to the respective unknowns $[a, Q]$. Simultaneously, all information on globals no longer protected, is *forgotten* to obtain the new local state. In this way, the analysis is fully relational in the local state, while only keeping relations within clusters of globals jointly protected by some mutex.

For clarity of presentation, we perform *control-point splitting* on the set of held mutexes when reaching program points. Apart from this, the constraint system and right-hand sides for the analysis closely follow those of the concrete semantics (Section 3) — with the exception that unknowns now take values from $\mathcal{R}$ and that unknowns $[a]$ are replaced with unknowns $[a, Q]$ for $Q \in \mathcal{Q}_a$.

All right-hand sides are given in detail in Fig. 4. For the **start point** of the program and the empty lockset, the right-hand side $\mathsf{init}^\sharp$ returns the $\top$ relation updated such that the variable self holds the abstract thread *id* i_0 of the *main*

$$
\begin{array}{l}
\mathsf{init}^\sharp\,\eta = \\
\quad \textbf{let } r(Q) = [\![\{g \leftarrow 0 \mid g \in Q\}]\!]^\sharp_{\mathcal{R}} \top \textbf{ in} \\
\quad \textbf{let } \rho = \{[a,Q] \mapsto r(Q) \mid a \in \mathsf{M}, Q \in \mathcal{Q}_a\} \\
\quad \textbf{in } (\rho, [\![\mathsf{self} \leftarrow^\sharp i_0]\!]^\sharp_{\mathcal{R}} \top) \\[1ex]
[\![[u,S], x{=}\mathsf{create}(u_1)]\!]^\sharp \eta = \\
\quad \textbf{let } r = \eta\,[u,S] \textbf{ in} \\
\quad \textbf{let } i = \nu^\sharp\, u\, u_1\, r \textbf{ in} \\
\quad \textbf{let } r' = \left\{ [\![\mathsf{self} \leftarrow^\sharp i]\!]^\sharp_{\mathcal{R}}\, r \right\} \Big|_{\mathcal{X}} \textbf{ in} \\
\quad \textbf{let } \rho = \{[u_1, \emptyset] \mapsto r'\} \textbf{ in} \\
\quad (\rho, [\![x \leftarrow^\sharp i]\!]^\sharp_{\mathcal{R}} r) \\[1ex]
[\![[u,S], g = x]\!]^\sharp \eta = \\
\quad (\emptyset, [\![g \leftarrow x]\!]^\sharp_{\mathcal{R}}\, (\eta\,[u,S])) \\[1ex]
[\![[u,S], x = g]\!]^\sharp \eta = \\
\quad (\emptyset, [\![x \leftarrow g]\!]^\sharp_{\mathcal{R}}\, (\eta\,[u,S]))
\end{array}
\qquad
\begin{array}{l}
[\![[u,S], \mathsf{lock}(a)]\!]^\sharp \eta = \\
\quad \left(\emptyset, \eta\,[u,S] \sqcap \left(\textstyle\bigsqcap_{Q \in \mathcal{Q}_a} \eta\,[a,Q]\right)\right) \\
[\![[u,S], \mathsf{unlock}(a)]\!]^\sharp \eta = \\
\quad \textbf{let } r = \eta\,[u,S] \textbf{ in} \\
\quad \textbf{let } \rho = \{[a,Q] \mapsto r|_Q \mid Q \in \mathcal{Q}_a\} \textbf{ in} \\
\quad \left(\rho, r|_{\mathcal{X} \cup \bigcup\{\mathcal{G}[a'] \mid a' \in (S \setminus a)\}}\right) \\
[\![[u,S], \mathsf{return}\; x]\!]^\sharp \eta = \\
\quad \textbf{let } r = \eta\,[u,S] \textbf{ in} \\
\quad \textbf{let } i^\sharp = \mathsf{unlift}\, r\, \mathsf{self} \textbf{ in} \\
\quad \left(\left\{ [i^\sharp] \mapsto \left([\![\mathsf{ret} \leftarrow x]\!]^\sharp_{\mathcal{R}}\, r\right) \Big|_{\{\mathsf{ret}\}} \right\}, r \right) \\
[\![[u,S], x'{=}\mathsf{join}(x)]\!]^\sharp \eta = \\
\quad \textbf{let } v = \bigsqcup_{\mathsf{unlift}\, r\, x' \sqcap i' \neq \bot} \mathsf{unlift}\, (\eta[i'])\, \mathsf{ret} \textbf{ in} \\
\quad \left(\emptyset, [\![x' \leftarrow^\sharp v]\!]^\sharp_{\mathcal{R}} (\eta\,[u,S])\right)
\end{array}
$$

Fig. 4: Right-hand sides for the basic analysis. All functions are strict in $\bot$ (describing the empty set of local traces), we only display definitions for non-$\bot$ abstract values here. $[\![\{g \leftarrow 0 \mid g \in Q\}]\!]^\sharp_{\mathcal{R}}$ is shorthand for the abstract transformer corresponding to the assignment of 0 to all variables in Q one-by-one.

thread. Additionally, $\mathsf{init}^\sharp$ produces a side-effect for each mutex a and cluster Q that initializes all globals from the cluster with the value 0.

For a **thread creating** edge starting in program point u with lockset S, the right-hand side $[\![[u,S], x{=}\mathsf{create}(u_1)]\!]^\sharp$ first generates a new abstract thread *id*, which we assume can be computed using function $\nu^\sharp$. The new *id* is assigned to the variable x in the local state of the current thread. Additionally, the start state r' for the newly created thread is constructed and side-effected to the thread's start point with empty lockset $[u_1, \emptyset]$. Since threads start with empty lockset, the state r' is obtained by removing all information about globals from the local state of the creator and assigning the new abstract thread *id* to the variable self.

When **locking** a mutex a, the states stored at unknowns $[a,Q]$ with $Q \in \mathcal{Q}_a$ are combined with the local state by *meet*. This is sound because the value stored at any $[a,Q]$ only maintains relationships between variables write-protected by a, and these values soundly account for the program state at every $\mathsf{unlock}(a)$ and at program start. When **unlocking** a, on the other hand, the local state restricted to the appropriate clusters $Q \in \mathcal{Q}_a$ is side-effected to the respective unknowns $[a,Q]$, so that the changes made to variables in the cluster become visible to other threads. Also, the local state is restricted to the local variables and only those globals for which at least one protecting mutex is still held.

As special mutexes m_g immediately surrounding accesses to g are used to ensure atomicity, and information about g is associated with them, all **reads** and **writes** refer to the local copy of g. **Guards** and **assignments** (which may only involve local variables) are defined analogously. For a **return** edge, the abstract value to be returned is looked up in the local state and then side-effected to the abstract thread *id* of the current thread (as the value of the dedicated variable ret). For **join**, the least upper bound of all *return* values of all possibly joined threads is assigned to the left-hand side of the *join* statement in the local state.

Example 4. Consider the program[4] where $\mathcal{M}[g] = \{a, b, m_g\}$, $\mathcal{M}[h] = \{a, b, m_h\}$, $\mathcal{Q}_a = \{\{g, h\}\}$, $\mathcal{Q}_b = \{\{g, h\}\}$.

```
main:                           t1:                        t2:
 x = create(t1); y = ?;          lock(b);                   lock(b);
 lock(a); lock(b);               unlock(b);                 lock(a);
 g = y; h = y+9;                 lock(a);                   // ASSERT(g==h); (4)
 unlock(b); lock(b);             lock(b);                   unlock(a);
 h = y;                          // ASSERT(g==h); (3)       unlock(b);
 // ASSERT(g==y); (1)            y = ?; g = y; h = y;
 // ASSERT(h==y); (2)            unlock(b);
 unlock(b); unlock(a);           unlock(a);
 x = create(t2);
```

Our analysis succeeds in proving all assertions here. Thread t_2 is of particular interest: When locking b only $g \leq h$ is known to hold, and locking the additional mutex a means that the better information $g = h$ becomes available. The analysis by Mukherjee et al. [42] on the other hand only succeeds in proving assertion (2) — even when all globals are put in the same region. It cannot establish (1) because all correlations between locals and globals are forgotten when the *mix* operation is applied at the second lock(b) in the main thread. (3) cannot be established because, at lock(b) in t_1, the mix operation also incorporates the state after the first unlock(b) in the main thread, where $g = h$ does not hold. Similarly, for (4). The same applies for assertion (3) and the analysis using *lock invariants* proposed by Miné [39]. This analysis also falls short of showing (1), as at the lock(b) in the main thread, the lock invariant associated with b is joined into the local state. (4) is similarly out of reach. The same reasoning also applies to [39, 42, 48] after equipping the analyses with thread *ids*. □

Theorem 1. *Any solution of the constraint system is sound w.r.t. the local trace semantics.*

Proof. The proof is by fixpoint induction, the details are given in Appendix B of the extended version [49] of this paper.

We remark that, instead of relying on $\mathcal{M}[g]$ being pre-computed, an analysis can also infer this information on the fly [58].

The analysis however still has some deficiencies. All writes to a global are accumulated regardless of the writing thread. As a consequence, a thread does, e.g., not only read its latest local writes but also all earlier local writes, even if

[4] In all examples, g, h, and i are globals, whereas x, y, and z are locals, and the clusters at special mutexes m_g contain only g: $\mathcal{Q}_{m_g} = \{\{g\}\}$. Unless explicitly stated otherwise, domain $\mathcal{R}_1$ from Example 2, enhanced with variable inequalities is used.

those are *definitely* overwritten. Excluding some threads' writes is an instance of the more general idea of excluding writes that cannot be last writes. Instead of giving ad hoc remedies for this specific shortcoming, we propose a general mechanism to improve the precision of any thread-modular analysis in the next section, and later instantiate it to the issue highlighted here.

5 Refinement via Finite Abstractions of Local Traces

To improve precision of thread-modular analyses we take additional abstractions of local traces into account. Our approach is generic, building on the right-hand sides of a base analysis and using them as black boxes. We will instantiate this framework to exclude writes based on thread *id*s from the analysis in Section 4. Other instantiations are conceivable as well. To make it widely applicable, the framework allows base analyses that already perform some splitting of unknowns at program points (e.g., locksets in Section 4). We denote by $[\hat{u}]$ such (possibly) extended unknowns for a program point u. A (base) analysis is defined by its right-hand sides, and a collection of domains: (1) $\mathcal{D}_S$ for abstract values stored at unknowns for program points; (2) $\mathcal{D}_{\mathsf{act}}$ for abstract values stored at observable actions act (e.g., in Section 4, $\mathcal{D}_M$ for unlocks and $\mathcal{D}_T$ for thread returns).

Let $\mathcal{A}$ be a set of finite information that can be extracted from a local trace by a function $\alpha_{\mathcal{A}}{:}\mathcal{T}{\to}\mathcal{A}$. We call $\alpha_{\mathcal{A}}\, t{\in}\mathcal{A}$ the *digest* of some local trace t. Let $[\![u, \mathsf{act}]\!]^\sharp_{\mathcal{A}}{:}\mathcal{A}^k{\to}2^{\mathcal{A}}$ be the effect on the digest when performing a k-ary action $\mathsf{act} \in \mathcal{A}ct$ for a control flow edge originating at u. We require for $e{=}(u, \mathsf{act}, v){\in}\mathcal{E}$,

$$\begin{array}{ll}\forall A_0,\ldots,A_{k-1} \in \mathcal{A}: & |[\![u, \mathsf{act}]\!]^\sharp_{\mathcal{A}}(A_0,\ldots,A_{k-1})| \leq 1 \\ \forall t_0,\ldots,t_{k-1} \in \mathcal{T} & : \alpha_{\mathcal{A}}([\![e]\!](t_0,\ldots,t_{k-1})) \subseteq [\![u, \mathsf{act}]\!]^\sharp_{\mathcal{A}}(\alpha_{\mathcal{A}}\, t_0,\ldots,\alpha_{\mathcal{A}}\, t_{k-1})\end{array} \tag{4}$$

where $\alpha_{\mathcal{A}}$ is lifted element-wise to sets. While the first restriction ensures determinism, the second intuitively ensures that $[\![u, \mathsf{act}]\!]^\sharp_{\mathcal{A}}$ soundly abstracts $[\![e]\!]$.

For thread creation, we additionally require a helper function $\mathsf{new}^\sharp_{\mathcal{A}} : \mathcal{N} \to \mathcal{N} \to \mathcal{A} \to \mathcal{A}$ that returns for a thread created at an edge originating from u and starting execution at program point u_1 the new digest. The same requirements are imposed for edges $(u, x{=}\mathsf{create}(u_1), v) \in \mathcal{E}$,

$$\forall A_0{\in}\mathcal{A} : |\mathsf{new}^\sharp_{\mathcal{A}}\, u\, u_1\, A_0| \leq 1 \qquad \forall t_0{\in}\mathcal{T} : \alpha_{\mathcal{A}}(\mathsf{new}\, u\, u_1\, t) \subseteq \mathsf{new}^\sharp_{\mathcal{A}}\, u\, u_1\, (\alpha_{\mathcal{A}}\, t_0) \tag{5}$$

Also, we define for the initial digest at the start of the program

$$\mathsf{init}^\sharp_{\mathcal{A}} = \{\alpha_A\, t \mid t \in \mathsf{init}\} \tag{6}$$

Under these assumptions, we can perform *control-point splitting* according to $\mathcal{A}$. This means that unknowns $[\hat{u}]$ for program points u are replaced with new unknowns $[\hat{u}, A]$, $A \in \mathcal{A}$. Analogously, unknowns for observable actions $[\mathsf{act}]$ are replaced with unknowns $[\mathsf{act}, A]$ for $A \in \mathcal{A}$. Consider a single constraint from an abstract constraint system of the last section, which soundly abstracts the collecting local trace semantics of a program.

$$(\eta, \eta\,[\hat{v}]) \sqsupseteq [\![[\hat{u}], \mathsf{act}]\!]^\sharp\, \eta$$

```
$[\![[\hat{u}, A_0], \mathsf{act}, A_1]\!]^\sharp\, \eta =$
  $\textbf{let}\, \eta'\, [x] = \textbf{if}\, [x] = [\hat{u}]\, \textbf{then}$
      $\eta\, [\hat{u}, A_0]$
    $\textbf{else}\, \eta\, [x, A_1]$
  $\textbf{in}$
  $[\![[\hat{u}], \mathsf{act}]\!]^\sharp\, \eta'$

$[\![[\hat{u}, A_0], \mathsf{act}'']\!]^\sharp\, \eta =$
  $\textbf{let}\, \eta'\, [x] = \eta\, [x, A_0]\, \textbf{in}$
  $[\![[\hat{u}], \mathsf{act}'']\!]^\sharp\, \eta'$

$[\![[\hat{u}, A_0], \mathsf{act}', A']\!]^\sharp\, \eta =$
  $\textbf{let}\, \eta'\, [x] = \eta\, [x, A_0]\, \textbf{in}$
  $\textbf{let}\, (\rho, v) = [\![[\hat{u}], \mathsf{act}']\!]^\sharp\, \eta'\, \textbf{in}$
  $\textbf{let}\, \rho' = \{[x, A'] \mapsto v' \mid ([x] \mapsto v') \in \rho\}\, \textbf{in}$
  $(\rho', v)$

$[\![[\hat{u}, A_0], x{=}\mathsf{create}(u_1)]\!]^\sharp\, \eta =$
  $\textbf{let}\, \eta'\, [x] = \eta\, [x, A_0]\, \textbf{in}$
  $\textbf{let}\, (\{[\hat{u_1}] \mapsto v'\}, v) = [\![[\hat{u}], x{=}\mathsf{create}(u_1)]\!]^\sharp\, \eta'\, \textbf{in}$
  $(\{[\hat{u_1}, A'] \mapsto v' \mid A' \in \mathsf{new}^\sharp_{\mathcal{A}}\, u\, u_1\, A_0\}, v)$
```

Fig. 5: Right-hand sides for an observing action act, an observable action act', a create action, and an action act'' that is neither for the refined analyses, defined as wrappers around the right-hand sides of a base analysis.

The corresponding constraints of the refined system with control-point splitting differ based on whether the action act is observing, observable, or neither.

– When act is *observing*, the new right-hand side additionally gets the digest A_1 associated with the local traces that are to be incorporated:

$$(\eta, \eta\ [\hat{v}, A']) \sqsupseteq [\![[\hat{u}, A_0], \mathsf{act}, A_1]\!]^\sharp\, \eta \qquad \text{for } A_0, A_1 \in \mathcal{A}, A' \in [\![u, \mathsf{act}]\!]^\sharp_{\mathcal{A}}\, (A_0, A_1)$$

– When act is *observable*, the digest A' of the resulting local trace is passed, so the side-effect can be redirected to the appropriate unknown:

$$(\eta, \eta\ [\hat{v}, A']) \sqsupseteq [\![[\hat{u}, A_0], \mathsf{act}, A']\!]^\sharp\, \eta \qquad \text{for } A_0 \in \mathcal{A}, A' \in [\![u, \mathsf{act}]\!]^\sharp_{\mathcal{A}}\, (A_0)$$

– When act is neither, no additional digest is passed:

$$(\eta, \eta\ [\hat{v}, A']) \sqsupseteq [\![[\hat{u}, A_0], \mathsf{act}]\!]^\sharp\, \eta \qquad \text{for } A_0 \in \mathcal{A}, A' \in [\![u, \mathsf{act}]\!]^\sharp_{\mathcal{A}}\, (A_0)$$

The new right-hand sides are defined in terms of the right-hand side of the base analysis which are used as black boxes (Fig. 5). They act as wrappers, mapping any unknown consulted or side-effected to by the original analysis to the appropriate unknown of the refined system. Thus, the refined analysis automatically benefits from the extra information the digests provide. It may, e.g., exploit that $[\![u, \mathsf{act}]\!]^\sharp_{\mathcal{A}}(A_0, A_1) = \emptyset$ meaning that, no local traces with digests A_0, A_1 can be combined into a valid local trace ending with action act. The complete definition of the refined constraint system instantiated to the actions considered here and unknowns for program points enriched with locksets is given in [49, Fig. 14].

Enriching program points with locksets can in fact be seen as a first application of this framework. The right-hand sides are given in Fig. 6.

Example 5. As a further instance, consider tracking which mutexes have been locked at least once in the local trace. At $\mathsf{lock}(a)$ traces in which a thread has performed a $\mathsf{lock}(a)$ can not be combined with traces that contain no $\mathsf{lock}(a)$. The corresponding right-hand sides are given in Fig. 7. When refining the analysis from Section 4 accordingly (assuming a protects g and h), it succeeds in proving the assert in this program as the initial values of 0 for g and h can be excluded.

```
main:                    t1:                      t2:
 lock(a);                 x = create(t2);          lock(a);
 h = 9; g = 10;           lock(a);                 // ASSERT(h<=g);
 unlock(a);               h = 11; g = 12;          unlock(a);
 x = create(t1);          unlock(a);
```

This naturally generalizes to counting how often some action (e.g., a write to a global g) occurred, stopping exact bookkeeping at a constant (1 in this case). □

To prove soundness of local-trace-based refinement of our analysis from Section 4, we first construct a corresponding refined collecting local trace semantics. Then we verify that the refined analysis is sound w.r.t. this refined semantics – which, in turn, is proven sound w.r.t. the original collecting local trace semantics.

Theorem 2. *Assume that $\alpha_\mathcal{A}$, $\mathsf{new}^\sharp_\mathcal{A}$, and $[\![u, \mathsf{act}]\!]^\sharp_\mathcal{A}$ fulfill requirements (4), (5), and (6). Then any solution of the refined constraint system is sound relative to the collecting local trace semantics.*

Proof. A proof sketch instantiated with the actions considered here and unknowns enriched with locksets is provided in [49, Appendix D].

6 Analysis of Thread Ids and Uniqueness

We instantiate the scheme from the previous section to compute abstract thread *id*s and their uniqueness. That refinement of the base analysis enhances precision of the analysis by excluding reads, e.g., from threads that have not yet been started. For that, we identify threads by their thread creation history, i.e., by sequences of *create* edges. As these sequences may grow arbitrarily, we collect all creates occurring after the first repetition into a *set* to obtain finite abstractions.

Example 6. In the program from Fig. 8, the first thread created by *main* receives the abstract thread *id* $(\mathsf{main} \cdot \langle u_1, t_1\rangle, \emptyset)$. It creates a thread with abstract thread *id* $(\mathsf{main} \cdot \langle u_1, t_1\rangle \cdot \langle u_3, t_1\rangle, \emptyset)$. At program point u_3, the latter creates a thread starting at t_1 and receiving the abstract thread *id* $(\mathsf{main} \cdot \langle u_1, t_1\rangle, \{\langle u_3, t_1\rangle\})$ – as do all threads subsequently created at this edge. □

$$\begin{array}{ll}
\mathsf{init}^\sharp_\mathcal{A} = \{\emptyset\} & [\![u, \mathsf{lock}(a)]\!]^\sharp_\mathcal{A}\,(S, S') = \{S \cup \{a\}\} \\
\mathsf{new}^\sharp_\mathcal{A}\, u\, u_1\, S = \{\emptyset\} & [\![u, \mathsf{unlock}(a)]\!]^\sharp_\mathcal{A}\, S = \{S \setminus \{a\}\} \\
[\![u, a]\!]^\sharp_\mathcal{A}\, S = \{S\} \quad \text{(other non-observing)} & [\![u, a]\!]^\sharp_\mathcal{A}\,(S, S') = \{S\} \quad \text{(other observing)}
\end{array}$$

Fig. 6: Right-hand sides for expressing locksets as a refinement.

Create edges, however, may also be repeatedly encountered within the creating thread, in a loop. To deal with this, we track for each thread, the set C of possibly already encountered *create* edges. As soon as a *create* edge is encountered again, the created thread receives a non-unique thread *id*.

Example 7. The first time the main thread reaches program point u_2 in the program from Fig. 8, the created thread is assigned the *unique* abstract thread *id* $(\mathsf{main} \cdot \langle u_2, t_1\rangle, \emptyset)$. In subsequent loop iterations, the created threads are no longer kept separate, and thus receive the non-unique *id* $(\mathsf{main}, \{\langle u_2, t_1\rangle\})$. $\square$

Formally, let $\mathcal{N}_C, \mathcal{N}_S$ denote the subsets of program points with outgoing edge labeled x=$\mathsf{create}(...)$, and of starting points of threads, respectively. Let $\mathcal{P} \subseteq \mathcal{N}_\mathcal{C} \times \mathcal{N}_\mathcal{S}$ denote sets of pairs relating thread creation nodes with the starting points of the created threads. The set $\mathcal{I}^\sharp$ of abstract thread *id*s then consists of all pairs $(i, s) \in (\mathsf{main} \cdot \mathcal{P}^*) \times 2^\mathcal{P}$ in which each pair $\langle u, f\rangle$ occurs at most once. Given the set $\mathcal{I}^\sharp$, we require that there is a concretization $\gamma : \mathcal{I}^\sharp \to 2^\mathcal{I}$ and a function $\mathsf{single} : \mathcal{I}^\sharp \to \mathcal{V}^\sharp_\mathcal{I}$ with $\gamma\, i^\sharp \subseteq \gamma_{\mathcal{V}^\sharp}\, (\mathsf{single}\, i^\sharp)$. The abstract thread *id* of the *main* thread is given by $(\mathsf{main}, \emptyset)$. Therein, the elements in $(\mathsf{main} \cdot \mathcal{P}^*) \times \{\emptyset\}$ represent the *unique* thread *id*s representing at most one concrete thread *id*, while the elements (i, s), $s \neq \emptyset$, are *ambiguous*, i.e., may represent multiple concrete thread *id*s. Moreover, we maintain the understanding that the concretizations of distinct abstract thread *id*s from $\mathcal{I}^\sharp$ all are disjoint.

As refining information $\mathcal{A}$ we consider not only abstract thread *id*s – but additionally track sets of executed thread creations within the current thread. Accordingly, we set $\mathcal{A} = \mathcal{I}^\sharp \times 2^P$ and define the right-hand sides as seen in Fig. 9, where $\bar{i}$ denotes the set of pairs occurring in the sequence i.

Example 8. Consider again the program from Fig. 8 with right-hand sides from Fig. 9, and assume that the missing right-hand for join returns its first argument. The initial thread has the abstract thread *id* $i_0 = (\mathsf{main}, \emptyset)$. At its start point, the digest thus is $(i_0, \emptyset)$. At the create edge originating at u_1, a new thread with *id* $(\mathsf{main} \cdot \langle u_1, t_1\rangle, \emptyset)$ is created. The digest for this thread then is $((\mathsf{main} \cdot \langle u_1, t_1\rangle, \emptyset), \emptyset)$. For the main thread, the encountered create edge $\langle u_1, t_1\rangle$ is added to the second component of the digest, making it $(i_0, \{\langle u_1, t_1\rangle\})$.

When u_2 is reached with $(i_0, \{\langle u_1, t_1\rangle\})$, a unique thread with *id* $(\mathsf{main} \cdot \langle u_2, t_1\rangle, \emptyset)$ is created. The new digest of the creating thread then is $(i_0, \{\langle u_1, t_1\rangle, \langle u_2, t_1\rangle\})$. In subsequent iterations of the loop, for which u_2 is reached with $(i_0, \{\langle u_1, t_1\rangle, \langle u_2, t_1\rangle\})$, a non-unique thread with *id* $(\mathsf{main}, \{\langle u_2, t_1\rangle\})$ is created.

$$\mathsf{init}^\sharp_\mathcal{A} = \{\emptyset\} \qquad \mathsf{new}^\sharp_\mathcal{A}\, u\, u_1\, L = \{L\}$$

$$\llbracket u, \mathsf{lock}(a)\rrbracket^\sharp_\mathcal{A}\, (L, L') = \begin{cases} \emptyset & \text{if } a \in L \wedge a \notin L' \\ \{L \cup L' \cup \{a\}\} & \text{otherwise} \end{cases}$$

$$\llbracket u, a\rrbracket^\sharp_\mathcal{A}\, S = \{L\} \ \ \text{(other non-observing)} \qquad \llbracket u, a\rrbracket^\sharp_\mathcal{A}\, (L, L') = \{L \cup L'\} \ \ \text{(other observing)}$$

Fig. 7: Right-hand sides for refining according to encountered lock operations.

When reaching u_3 with *id* $(\mathsf{main}, \{\langle u_2, t_1\rangle\})$, a thread with *id* $(\mathsf{main}, \{\langle u_2, t_1\rangle, \langle u_3, t_1\rangle\})$ is created as the *id* of the creating thread was already not unique. When reaching it with the *id* $(\mathsf{main} \cdot \langle u_1, t_1\rangle, \emptyset)$, a new thread with *id* $(\mathsf{main} \cdot \langle u_1, t_1\rangle \cdot \langle u_3, t_1\rangle, \emptyset)$ is created. When the newly created thread reaches this program point, the threads created there have the *non-unique id* $(\mathsf{main} \cdot \langle u_1, t_1\rangle, \{\langle u_3, t_1\rangle\})$, as $\langle u_3, t_1\rangle$ already appears in the *id* of the creating thread. □

Abstract thread *id*s should provide us with functions

– $\mathsf{unique} : \mathcal{I}^\sharp \to \mathbf{bool}$ tells whether a thread *id* is unique.
– $\mathsf{lcu\_anc} : \mathcal{I}^\sharp \to \mathcal{I}^\sharp \to \mathcal{I}^\sharp$ returns the last common *unique* ancestor of two threads.
– $\mathsf{may\_create} : \mathcal{I}^\sharp \to \mathcal{I}^\sharp \to \mathbf{bool}$ checks whether a thread *may* (transitively) create another.

For our domain $\mathcal{I}^\sharp$, these can be defined as $\mathsf{unique}\,(i, s) = (s = \emptyset)$ and

$$\begin{aligned}\mathsf{lcu\_anc}\,(i,s)\,(i',s') &= (\mathsf{longest\ common\ prefix}\ i\,i', \emptyset)\\ \mathsf{may\_create}\,(i,s)\,(i',s') &= (\bar{i} \cup s) \subseteq (\bar{i'} \cup s')\end{aligned}$$

We use this extra information to enhance the definitions of $[\![u, \mathsf{lock}(a)]\!]^\sharp_\mathcal{A}$ and $[\![u, x'{=}\mathsf{join}(x)]\!]^\sharp_\mathcal{A}$ to take into account that the ego thread cannot acquire a mutex from another thread or join a thread that has definitely not yet been created. This is the case for a thread t'

(1) that is directly created by the *unique* ego thread, but the ego thread has not yet reached the program point where t' is created;
(2) whose thread *id* indicates that a thread that has not yet been created according to (1), is part of the creation history of t'.

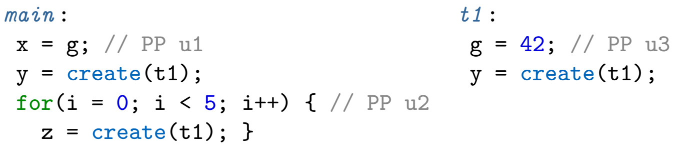
Accordingly, we introduce the predicate $\mathsf{may\_run}\,(i, C)\,(i', C')$ defined as

$$(\mathsf{lcu\_anc}\, i\, i' = i) \implies \exists \langle u, u'\rangle \in C : (i \circ \langle u, u'\rangle = i' \vee \mathsf{may\_create}\,(i \circ \langle u, u'\rangle)\, i')$$

which is false whenever thread i' is definitely not yet started. We then set

$$\begin{aligned}[\![u, \mathsf{lock}(a)]\!]^\sharp_\mathcal{A}\,(i,C)\,(i',C') &= [\![u, x'{=}\mathsf{join}(x)]\!]^\sharp_\mathcal{A}\,(i,C)\,(i',C')\\ &= \begin{cases}\{(i,C)\} & \text{if } \mathsf{may\_run}\,(i,C)\,(i',C')\\ \emptyset & \text{otherwise}\end{cases}\end{aligned}$$

This analysis of thread *id*s and uniqueness can be considered as a *May-Happen-In-Parallel* (or, more precisely, *Must-Not-Happen-In-Parallel*) analysis. MHP



```
main:                                   t1:
 x = g; // PP u1                         g = 42; // PP u3
 y = create(t1);                         y = create(t1);
 for(i = 0; i < 5; i++) { // PP u2
   z = create(t1); }
```

Fig. 8: Program with multiple thread creations.

information is useful in a variety of scenarios: a thread-modular analysis of *data races* or *deadlocks*, e.g., that does not consider thread *ids* and joining, can be refined with this analysis to exclude more data races or deadlocks. Subsequently, we outline how the analysis from Section 4 may benefit from MHP information.

7 Exploiting Thread *IDs* to Improve Relational Analyses

We subsequently exploit abstract thread *ids* and their uniqueness to limit the amount of reading performed by the analysis from Section 4.

I1 from other threads that have not yet been created.
I2 the ego thread's past writes, if its thread *id* is unique.
I3 past writes from threads that have already been joined.

Improvements **I1** and **I3** have, e.g., been realized in a setting where thread *ids* and which thread is joined where can be read off from control-flow graphs [31]. Here, however, this information is computed *during* analysis. In our framework, **I1** is already achieved by refining the base analysis according to Section 6.

Example 9. Consider the program below where $\mathcal{M}[g] = \{a, b, m_g\}$, $\mathcal{M}[h] = \{a, b, m_h\}$, $\mathcal{M}[i] = \{m_i\}$ and assume $\mathcal{Q}_a = \{\{g, h\}\}$.

```
main:                                    t1:
 x = create(t1); lock(a);                 lock(a);
 // ASSERT(g==h);  (1)                     r = ?; g = r; h = r;
 unlock(a);                               unlock(a);
 y = create(t2); lock(a);
 // ASSERT(g==h);  (2)                   t2:
 g = 42; h = 42;                          lock(a); v = g; unlock(a);
 unlock(a); z = create(t3);
 i = 3;  i = 2; // ASSERT(i==2); (3)     t3:
 i = 8;                                   lock(a); g = 19; unlock(a);
```

The analysis succeeds in proving (1), as the thread (starting at) t_3 that breaks the invariant $g{=}h$ has definitely not been started yet at this program point. Without refinement, the analysis from Section 4 could not prove (1). However, this does

```
$\mathsf{init}^\sharp_{\mathcal{A}} = \{((\mathsf{main}, \emptyset), \emptyset)\}$
$[\![u, x{=}\mathsf{create}(u_1)]\!]^\sharp_{\mathcal{A}}\,(i, C) = \{(i, C \cup \{\langle u, u_1\rangle\})\}$
$[\![u, a]\!]^\sharp_{\mathcal{A}}\,(i, C) = \{(i, C)\}$        (for other actions $a$)
$\mathsf{new}^\sharp_{\mathcal{A}}\; u\, u_1\; ((d, s), C) =$
    let $(d', s') = (d, s) \circ \langle u, u_1\rangle$ in
    if $s' = \emptyset \wedge \langle u, u_1\rangle \in C$ then $((d, \{\langle u, u_1\rangle\}), \emptyset)$
    else $((d', s'), \emptyset)$

$(d, s) \circ \langle u, u_1\rangle =$
    if $d = (d_0 \cdot \langle u, u_1\rangle) \cdot d_1$ then
        $(d_0, s \cup \bar{d_1} \cup \{\langle u, u_1\rangle\})$
    else if $s = \emptyset$ then $(d \cdot \langle u, u_1\rangle, \emptyset)$
    else $(d, s \cup \{\langle u, u_1\rangle\})$
```

Fig. 9: Right-hand sides for thread *ids*.

not suffice to prove (2). At this program point, t_2 may already be started. At the $\mathsf{lock}(a)$ in t_2, t_3 may also be started; thus, the violation of the invariant $g{=}h$ by t_3 is incorporated into the local state of t_2 at lock. At $\mathsf{unlock}(a)$, despite t_2 *only reading* g, the imprecise abstract relation violating $g{=}h$, is side-effected to $[a, \{g, h\}, t_2]$ and is incorporated at the second $\mathsf{lock}(a)$ of the main thread. The final shortcoming is that each thread reads all its own past (and future!) writes – even when it is known to be unique. This means that (3) cannot be proven. □

To achieve **I2**, some effort is required as our analysis forgets values of globals when they become unprotected. This is in contrast, e.g., to [39, 42]. We thus restrict side-effecting to mutexes to cases where the ego thread has possibly written a protected global since acquiring it. This is in contrast to Section 4, where a side-effect is performed at every unlock, i.e., everything a thread reads is treated as if it was written by that thread.

Technically, we locally track a map $L : (\mathsf{M} \times \mathcal{Q}) \to \mathcal{R}$, where $L\,(a, Q)$ maintains for a mutex a, an abstract relation between the globals in cluster $Q \in \mathcal{Q}_a$. More specifically, the abstract relation on the globals from Q recorded in $L\,(a, Q)$ is the one that held when a was unlocked *join-locally* for the *first* time after the *last join-local* write to a global in $\mathcal{G}\,[a]$. If there is no such $\mathsf{unlock}(a)$, the relation at program start is recorded. We call an operation in a local trace *join-local* to the ego thread, if it is (a) *thread-local*, i.e., performed by the ego thread, or (b) is executed by a thread that is (transitively) joined into the ego thread, or (c) is *join-local* to the parent thread at the node at which the ego thread is created. This notion will also be crucial for realizing **I3**. *Join-locality* is illustrated in Fig. 10, where the *join-local* part of a local trace is highlighted.

For *join-local* contributions, it suffices to consult $L\,a$ instead of unknowns $[a, Q, i]$. Such contributions are *accounted* for. To check whether a contribution from some thread *id* is *accounted* for, we introduce a function $\mathsf{acc} : (\mathcal{A} \times \mathcal{D}_S) {\to} \mathcal{A} {\to} \mathsf{bool}$ (see definition (7) below). Besides an abstract value from $\mathcal{R}$, the local state $\mathcal{D}_S$ now contains two additional components:

– The map $L : (\mathsf{M} \times \mathcal{Q}) \to \mathcal{R}$ for which the join is given component-wise;
– The set $W : 2^{\mathcal{G}}$ (ordered by $\subseteq$) of globals that may have been written since one of its protecting mutexes has been locked, and not all protecting mutexes have been unlocked since.

Just like r, L and W are abstractions of the reaching local traces. $\mathcal{D}_T$ is also enhanced with an L component, while $\mathcal{D}_M$ remains unmodified. We sketch the right-hand sides here, definitions are given in Fig. 11. For **program start** $\mathsf{init}^\sharp$, in contrast to the analysis from Section 4, there is no initial side-effect to the unknowns for mutexes. The initial values of globals are *join-local*, and thus accounted for in the L component also passed to any subsequently created thread.

The right-hand sides for **thread creation** and **return** differ from the analysis from Section 4 enhanced with thread *id*s only in the handling of additional data structures L and W. As the thread *id*s are tracked precisely in the $\mathcal{A}$ component, this information is directly used when determining which unknown to side-effect to and unknowns $[(i, C)]$ replace unknowns $[i', (i, C)]$.

For **join**, if the return value of the thread is not accounted for, it is assigned to the variable on the left-hand side and the L information from the ego thread and the joined thread is joined. If, on the other hand, it is accounted for, the thread has already been joined and cannot be joined again. There is a separate constraint for each (i', C'), so that all threads that could be joined are considered.

For **locking** of mutexes, upon lock, if (i', C') is not accounted for, its information on the globals protected by a is joined with the *join-local* information for a maintained in $L\,(a, Q)$, $Q \in \mathcal{Q}_a$. This information about the globals protected by a is then incorporated into the local state by $\sqcap$. For **unlocking** of mutexes, if there may have been a write to a protected global since the mutex was locked (according to W), the *join-local* information is updated and the local state restricted to Q is side-effected to the appropriate unknown $[a, Q, (i, C)]$ for $Q \in \mathcal{Q}_a$. Just like in Section 4, r is then restricted to only maintain relationships between locals and those globals for which at least one protecting mutex is still held. **Reading** from and **writing** to globals once more are purely local operations. To exclude self writes, we set

$$\mathsf{acc}\,((i, C), \_)\,(i', C') = \mathsf{unique}\, i \wedge i = i' \tag{7}$$

The resulting analysis thus takes **I1** (via $[\![...]\!]^\sharp_{\mathcal{A}}$ defined in Section 6), as well as **I2** (via acc) into account. In Example 9, it is now able to show all assertions.

Theorem 3. *This analysis is sound w.r.t. to the local trace semantics.*

Proof. The proof relies on the following observations:

- When $\mathcal{G}[a] \cap W = \emptyset$, no side-effect is required.
- Exclusions based on acc are sound, i.e., it only excludes *join-local* writes.

The detailed proof is a simplification of a proof for an enhanced analysis from the extended version [49, Appendix F], which we outline in Appendix G there. $\square$

The analysis does not make use of components C at unknowns $[a, Q, (i, C)]$ and $[i, C]$. In [49, Appendix E], we detail how this information can be exploited to exclude a further class of writes – namely, those that are performed by an ancestor of the ego thread before the ego thread was created. Alternatively, an implementation may abandon control-point splitting according to C at mutexes and thread *ids*, replacing $[a, Q, (i, C)], [i, C]$ with $[a, Q, i]$ and $[i]$, respectively.

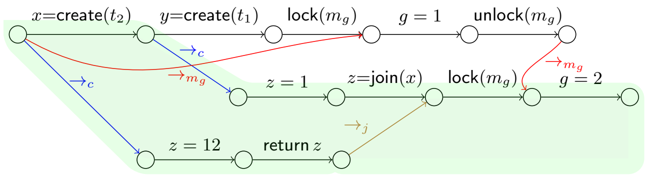


Fig. 10: Illustration highlighting the *join-local* part of a local trace of the program from Fig. 2a, and which writes are thus accounted for by L.

```
$\mathsf{init}^\sharp_{(i,C)} =$
    $\textbf{let}\ L\,(a,Q) = [\![\{g \leftarrow 0 \mid g \in Q\}]\!]^\sharp_{\mathcal{R}} \top\ \textbf{in}$
    $\textbf{let}\ r = [\![\mathsf{self} \leftarrow^\sharp i]\!]^\sharp_{\mathcal{R}} \top\ \textbf{in}$
    $(\emptyset, (\{(a,Q) \mapsto L\,(a,Q) \mid a{\in}\mathsf{M}, Q{\in}\mathcal{Q}_a\}, \emptyset, r))$

$[\![[u,S,(i,C)], x' = \mathsf{join}(x), (i',C')]\!]^\sharp\eta =$
    $\textbf{let}\ (L,W,r) = \eta\,[u,S,(i,C)]\ \textbf{in}$
    $\textbf{if}\ (\mathsf{single}\,i' \sqcap ((\mathsf{unlift}\,r)\,x){=}\bot)\ \textbf{then}$
     $\bot\ \textbf{elseif}\ \mathsf{acc}\,((i,C),(L,W,r))\,(i',C')$
    $\textbf{then}\ \bot\ \textbf{else}$
    $\textbf{let}\ (L',v) = \eta[(i',C')]\ \textbf{in}$
    $\textbf{let}\ r' = [\![x' \leftarrow^\sharp (\mathsf{unlift}\,v)\,\mathsf{ret}]\!]^\sharp_{\mathcal{R}} r\ \textbf{in}$
    $(\emptyset, (L \sqcup L', W, r'))$

$[\![[u,S,(i,C)], \mathsf{lock}(a), (i',C')]\!]^\sharp\eta =$
    $\textbf{let}\ (L,W,r) = \eta\,[u,S,(i,C)]\ \textbf{in}$
    $\textbf{let}\ r' = \textbf{if}\ \mathsf{acc}\,((i,C),(L,W,r))\,(i',C')$
     $\textbf{then}\ \bot\ \textbf{else} \bigsqcap_{Q\in\mathcal{Q}_a} \eta\,[a,Q,(i',C')]\ \textbf{in}$
    $\left(\emptyset, \left(L, W, r \sqcap \left(\left(\bigsqcap_{Q\in\mathcal{Q}_a} L\,(a,Q)\right) \sqcup r'\right)\right)\right)$

$[\![[u,S,(i,C)], g = x]\!]^\sharp\eta =$
    $\textbf{let}\ (L,W,r) = \eta\,[u,S,A]\ \textbf{in}$
    $(\emptyset, (L, W \cup \{g\}, [\![g \leftarrow x]\!]^\sharp_{\mathcal{R}}\, r))$

$[\![[u,S,(i,C)], x = g]\!]^\sharp\eta =$
    $\textbf{let}\ (L,W,r) = \eta\,[u,S,A]\ \textbf{in}$
    $(\emptyset, (L, W, [\![x \leftarrow g]\!]^\sharp_{\mathcal{R}}\, r))$

$[\![[u,S,(i,C)], x{=}\mathsf{create}(u_1)]\!]^\sharp\eta =$
    $\textbf{let}\ (L,W,r) = \eta\,[u,S,(i,C)]\ \textbf{in}$
    $\textbf{let}\ (i',C') = \mathsf{new}^\sharp_{\mathcal{A}}\, u\, u_1\, (i,C)\ \textbf{in}$
    $\textbf{let}\ r' = ([\![\mathsf{self} \leftarrow^\sharp (\mathsf{single}\,i')]\!]^\sharp_{\mathcal{R}} r)\big|_{\mathcal{X}}\ \textbf{in}$
    $\textbf{let}\ \rho{=}\{[u_1,(\emptyset,(i',C'))]{\mapsto}(L,\emptyset,r')\}\ \textbf{in}$
    $(\rho, (L, W, [\![x \leftarrow^\sharp \mathsf{single}\,i']\!]^\sharp_{\mathcal{R}} r))$

$[\![[u,S,(i,C)], \mathsf{return}\,x, (i,C)]\!]^\sharp\eta =$
    $\textbf{let}\ (L,W,r) = \eta\,[u,S,(i,C)]\ \textbf{in}$
    $\textbf{let}\ v = \left([\![\mathsf{ret} \leftarrow x]\!]^\sharp_{\mathcal{R}}\, r\right)\Big|_{\{\mathsf{ret}\}}\ \textbf{in}$
    $\textbf{let}\ \rho = \{[(i,C)] \mapsto (L,v)\}\ \textbf{in}$
    $(\rho, (L,W,r))$

$[\![[u,S,(i,C)], \mathsf{unlock}(a), (i,C)]\!]^\sharp\eta =$
    $\textbf{let}\ (L,W,r) = \eta\,[u,S,(i,C)]\ \textbf{in}$
    $\textbf{let}\ (L',\rho) = \textbf{if}\ \mathcal{G}[a]{\cap}W{=}\emptyset\ \textbf{then}\ (L,\emptyset)$
        $\textbf{else}\ (L \oplus \{(a,Q) \mapsto r|_Q \mid Q \in \mathcal{Q}_a\},$
            $\{[a,Q,(i,C))] \mapsto r|_Q \mid Q \in \mathcal{Q}_a\})$
    $\textbf{in}$
    $\textbf{let}\ r' = r|_{\mathcal{X} \cup \bigcup\{\mathcal{G}[a'] \mid a' \in (S \setminus a)\}}\ \textbf{in}$
    $\textbf{let}\ W'{=}\{W \mid g{\in}W, \mathcal{M}[g] \cap S\setminus\{a\}{\neq}\emptyset\}$
    $\textbf{in}\ (\rho, (L',W',r'))$
```

Fig. 11: Right-hand sides for the improved (**I1**, **I2**) analysis using thread *id*s.

When turning to improvement **I3**, we observe that after joining a thread t with a *unique* thread *id*, t cannot perform further writes. As all writes of joined threads are *join-local* to the ego thread, it is not necessary to read from the corresponding global unknowns. We therefore enhance the analysis to also track in the local state, the set J of thread *id*s for which join has definitely been called in the *join-local* part of the local trace and refine acc to take J into account:

$$\mathsf{acc}\,((i,C),(J,L,W,r))\,(i',C') = \mathsf{unique}\,i' \wedge (i = i' \vee i' \in J)$$

The extended version [49, Appendix F] gives details on this enhancement.

8 Exploiting Clustered Relational Domains

Naïvely, one might assume that tracking relations among a larger set of globals is necessarily more precise than between smaller sets. Interestingly, this is no longer true for our analyses, e.g., in presence of thread *id*s. A similar effect where relating more globals can deteriorate precision has also been observed in the context of an analysis using a data-flow graph to model interferences [19].

Example 10. Consider again Example 1 in the introduction with $\mathcal{Q}_a = \{\{g, h, i\}\}$. For this program, the constraint system of the analysis has a unique least solution. It verifies that assertion (1) holds. It assures for $[a, \{g, h, i\}, t_1]$ that $h{=}i$ holds, while for the *main* thread and the program point before each assertion, $L\,(a, \{g, h, i\}) = \{g{=}h, h{=}i\}$ holds, while for $[a, \{g, h, i\}, \mathsf{main}]$ and $[a, \{g, h, i\}, t_2]$ only $\top$ is recorded, as is for any relation associated with m_g, m_h, or m_i. Assertion (2), however, will not succeed, as the side-effect from t_1 causes the older values from the first write in the main thread to be propagated to the assertions as well, implying that while $h{=}i$ is proven, $g{=}h$ is not. □

Intuitively, the analysis loses precision because, at an unlock of mutex a, the current relationships between *all* clusters protected by a are side-effected. As soon as *one* global is written to, the analysis behaves as if *all* protected globals had been written. By limiting publishing to those clusters for which at least one global has been written, more precise information may remain at others.

In the improved analysis, when **unlocking** a mutex a, side-effects are only produced to clusters $Q \in \mathcal{Q}_a$ containing at least one global that was written to since the last $\mathsf{lock}(a)$. Definitions for locking and unlocking are given in Fig. 12.

For **locking** the mutex a, the abstract value to be incorporated into the local state is assembled from the contributions of different threads to the clusters. For that, the separate constraints for each admitted digest from Section 5 are combined into one for the set $\mathbf{I} = \{(i', C') \mid (i, C) \in [\![\mathsf{lock}(a)]\!]^\sharp_{\mathcal{A}}((i, C), (i', C'))\}$ of *all* admitted digests. This is necessary as side-effects to unaffected clusters at $\mathsf{unlock}(a)$ have been abandoned and thus the meet with the values for clusters of one thread at a time is unsound. For each Q, the join-local information $L\,(a, Q)$ is joined with all contributions to Q by threads that are not yet *accounted* for, but admitted for Q by the digests. Here, the contributions of threads that do *not* write Q is $\bot$, and thus do not affect the value for Q. Finally, the resulting value is used to improve the local state by *meet*. The right-hand side for $\mathsf{lock}(a)$ thus exploits the fine-grained, per-cluster MHP information provided by the digests and the predicate acc. We obtain:

Theorem 4. *Given domains $\mathcal{R}$ and $\mathcal{V}^\sharp$ fulfilling the requirements from Fig. 1, any solution of the constraint system is sound w.r.t. the local trace semantics. Maximum precision is obtained with $\mathcal{Q}_a = 2^{\mathcal{G}[a]}$.* □

For Example 1, with $\mathcal{Q}_a = 2^{\mathcal{G}[a]}$, both assertions are verified. Performing the analysis with all subclusters simultaneously can be expensive when sets $\mathcal{G}[a]$ are large. The choice of subclustering thus generally involves a trade-off between precision and runtime. This is different for k-decomposable relational domains:

Theorem 5. *Provided the relational domain is k-decomposable (Equation (2)), the clustered analysis using all subclusters of sizes at most k only, is equally precise as the clustered analysis using all subclusters $\mathcal{Q}_a = 2^{\mathcal{G}[a]}$ at mutexes a.*

Proof. Consider a solution η of the constraint system with $\mathcal{Q}_a = 2^{\mathcal{G}[a]}$. Then for unknowns $[a, Q, (i, C)]$ and $[a, Q', (i, C)]$ with $Q \subseteq Q'$ and $|Q| \leq k$, and values $r{=}\eta\,[a, Q, (i, C)]$, $r'{=}\eta\,[a, Q', (i, C)]$, we have that $r \sqsubseteq r'|_Q$ (whenever the smaller

```
$[\![ [u, S, (i, C)], \mathsf{unlock}(a), (i, C) ]\!]^\sharp \eta =$
  let $(L, W, r) = \eta\,[u, S, (i, C)]$ in
  let $\mathcal{Q}' = \{Q \mid Q \in \mathcal{Q}_a, Q \cap W \neq \emptyset\}$ in
  let $L' = L \oplus \{(a, Q) \mapsto r|_Q \mid Q \in \mathcal{Q}'\}$ in
  let $\rho = \{[a, Q, (i, C)] \mapsto r|_Q \mid Q \in \mathcal{Q}'\}$ in
  let $r' = r|_{\mathcal{X} \cup \bigcup\{\mathcal{G}[a'] \mid a' \in (S \setminus a)\}}$ in
  let $W' = \{W \mid g \in W, \mathcal{M}[g] \cap S \setminus \{a\} \neq \emptyset\}$ in
  $(\rho, (L', W', r''))$

$[\![ [u, S, (i, C)], \mathsf{lock}(a), \mathbf{I} ]\!]^\sharp \eta =$
  let $(L, W, r) = \eta\,[u, S, (i, C)]$ in
  let $l = ((i, C), (L, W, r))$ in
  let $J(Q) = \bigsqcup \{\eta\,[a, Q, (i', C')] \mid$
        $(i', C') \in \mathbf{I}, \neg\mathsf{acc}\; l\,(i', C')\}$ in
  let $r' = \bigsqcap_{Q \in \mathcal{Q}_a} (J(Q) \sqcup L\,(a, Q))$
  in
  $(\emptyset, (L, W, r \sqcap r'))$
```

Fig. 12: Right-hand sides for unlocking and locking when limiting side-effecting to potentially written clusters.

cluster receives a side-effect, so does the larger one). Thus, by k-decomposability, the additional larger clusters Q', do not improve the *meet* over the clusters of size at most k for individual thread *id*s as well as the *meet* of their *join*s over all thread *id*s. The same also applies to the clustered information stored in L. □

Example 11. Consider again Example 1. If the analysis is performed with clusters $\mathcal{Q}_a = \{\{h, i\}, \{g, h\}, \{g, i\}, \{g\}, \{i\}, \{h\}\}$ both assertions can be proven. □

The one element clusters, on the other hand, cannot be abandoned – as indicated by the example from Appendix H in the extended version [49].

9 Experimental Evaluation

We implemented [50] the analyses extending the context-sensitive static analyzer Goblint which provides the set of protecting mutexes for each global. The implementation tracks information about integral variables using either the *Interval* or the *Octagon* domains from Apron [29]. A comparison with other tools is difficult, for details see [49, Appendix I]:

- Duet [19] — Its benchmarks are only available as binary goto-programs which neither its current version nor any other tool considered here can consume. Since Duet does not support function calls, it could only be run on *some* of the benchmarks considered here.
- AstréeA [39] — A public version is available but not licensed for evaluation.
- Watts [31] — Since we were unable to run the tool on any program, we compared with the numbers reported by the authors.
- NR-Goblint [48] — Goblint with the non-relational analyses from [48].

We considered four different configurations, namely, *Interval:* the analysis from Section 4 with Intervals; *Octagon:* the same analysis with Octagons; *TIDs:* the analysis from Section 7 with enhancement [49, Appendix F] with Octagons; *Clusters: TIDs* using clusters of size at most 2 only. All benchmarks were run in

Table 2: Summary of evaluation results, with individual programs grouped together. For each group the number of programs and the total number of assertions are given. ✓ (✗) indicates that all (no) assertions are proven, otherwise the number of proven assertions is given. (—) indicates invalid results produced.

				Our analyzer					
Set	Group	#	Asserts	Interval (Sec. 4)	Octagon (Sec. 4)	TIDs (Sec. 7)	Clusters (Sec. 8)	NR-Goblint w/ interval	Duet
Our	Basic	3	4	✓	✓	✓	✓	✓	3
	Relational	10	35	✗	✓	✓	✓	4	2
	TID	12	19	✗	✗	✓	✓	✗	2
	Cluster	2	3	✗	✗	1	✓	✗	1
Goblint	POSIX	5	1679	1146	1490	✓	✓	1582	—
	SV-COMP	7	360	✓	✓	✓	✓	✓	—
Watts	Created	3	3	2	2	2	2	2	✗
	SV-COMP	5	5	1	1	1	1	1	✗
	LKMPG	1	2	✗	✗	✗	✗	✗	✗
	DDVerify	28	1071	1043	1043	✓	✓	1043	—
	Scalability	5	740	735	735	✓	✓	735	—
Ratcop		19	34	4	14	18	18	6	4

a virtual machine on an AMD EPYC 7742 64-Core processor[5] running Ubuntu 20.04. The results of our evaluation are summarized in Table 2.

Our benchmarks. To capture particular challenges for multi-threaded relational analysis, we collected a set of small benchmarks (including the examples from this paper) and added assertions. On these, we evaluated our analyzer, NR-Goblint, and Duet. Our analysis in the *Clusters* configuration is capable of verifying all the programs. The other tools could only prove a handful of relational assertions.

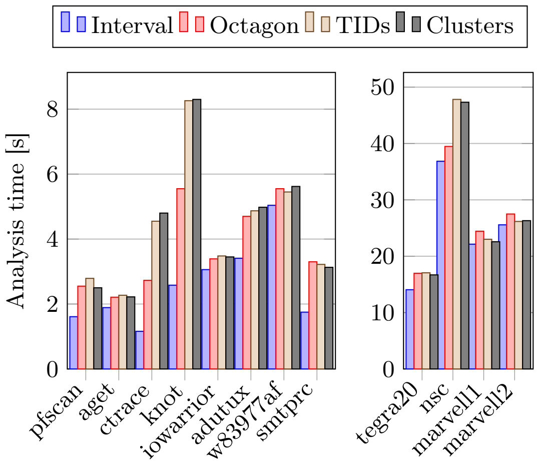
Goblint benchmarks [48]. These benchmarks do not contain assertions. To still relate the precision of our analyzer to the non-relational NR-Goblint and to Duet, we used our tool in the *Clusters* setting to automatically derive invariants at each locking operation. Perhaps surprisingly, NR-Goblint could verify 95% of the invariants despite being non-relational and not using thread *id*s.

Watts benchmarks [31]. These programs were instrumented with asserts and significantly changed by the authors. Our analyses can verify all but 7 out of over 1000 assertions. Due to necessary fixes to programs and our inability to run their tool, numbers are not directly comparable. Nevertheless, for their scalability tests, reported runtimes for Watts are up to two orders of magnitude worse than ours. See [49, Appendix I] for a more detailed discussion.

[5] The analyzer is single-threaded, so it only used one (virtual) core per analysis job.

Name	LLoC	#TIDs (unique)	TIDs ⊏ Octagon
pfscan	550	3 (2)	19.0%
aget	581	6 (4)	0.0%
ctrace	651	3 (3)	0.0%
knot	973	9 (5)	0.0%
smtprc	3013	2 (2)	0.8%
iowarrior	1358	4 (4)	17.1%
adutux	1509	4 (4)	0.0%
w83977af	1515	6 (4)	12.1%
tegra20	1560	7 (5)	0.0%
nsc	2394	11 (7)	32.2%
marvell1	2476	6 (5)	59.5%
marvell2	2476	6 (5)	58.4%

(a) Number of discovered thread *id*s and proportion of program points where analysis with thread *id*s is more precise.



(b) Analysis times.

Fig. 13: Precision and performance evaluation on the GOBLINT benchmark set.

RATCOP benchmarks [42]. These were JAVA programs. After manual translation to C, our analyzer succeeded in proving all assertions any configuration of RATCOP could with *Octagons*, while RATCOP required polyhedra in one case.

Internal comparison We evaluated our analyses in more detail on the GOBLINT benchmark set [48]. Fig. 13a shows sizes of the programs (in Logical LoC) and the number of thread *id*s found by the analysis from Section 6. The high number of threads identified as *unique* is encouraging. To evaluate precision, we compared the abstract values at each program point (joined over contexts). Fig. 13a shows for what proportion of program points tracking thread *id*s increases precision. There were no program points where precision decreased or values became incomparable, while for some programs gains of over 50% were observed. Fig. 13b illustrates runtimes. In 9 of 12 cases, performance differences between our relational analyses are negligible. In all cases, using clusters incurs no additional cost. Thus, the more precise analysis with clusters of size ≤ 2 seems to be the method of choice for thread-modular relational abstract interpretation.

10 Related Work

Since its introduction by Miné [36, 37], the weakly relational numerical domain of *Octagons* has found wide-spread application for the analysis and verification of programs [8, 14]. Since tracking relations between *all* variables may be expensive, pre-analyses have been suggested to identify *clusters* of numerical variables whose relationships may be of interest [8, 14, 26, 45]. A *dynamic* approach to decompose relational domains into non-overlapping clusters based on learning is proposed by Singh et al. [55]. While these approaches trade (unnecessary) precision for efficiency, others try to *partition* the variables into clusters without compromising precision [15, 23, 24, 44, 54, 56]. These types of clustering are orthogonal to our approach and could, perhaps, be combined with it.

The integration of relational domains into thread-modular abstract interpretation was pioneered by Miné [39]. His analysis is based on *lock invariants* determining for each mutex a relation which holds whenever the mutex is *not* held. *Weak interferences* are used to account for asynchronous variable accesses. For practical analyses, a relational abstraction only for lock invariants is proposed, while using a coarse, non-relational abstraction for the weak interferences. This framework closely follows the framework for non-relational analysis [38], while abandoning *background locksets*. Our relational analysis, on the other hand, maintains at each mutex a only relations between variables write-protected by a. For these relations more precise results can be obtained, since they are incorporated into the local state at locks by *meet* (while [39] uses *join*).

Monat and Miné [40] present an analysis framework which is orthogonal to our approach. It is tailored to the verification of algorithms that do not rely on explicit synchronization via mutexes such as the *Bakery* algorithm. Suzanne and Miné [57] extend [40] to handle weak memory effects (PSO, TSO) by incorporating memory buffers into the thread-local semantics. The notion of interferences is also used by Sharma and Sharma [52] for the analysis of programs under the Release/Acquire Memory Model of C11 by additionally tracking abstractions of *modification sequences* for global variables. They consider fixed finite sets of threads only, and do not deal with thread creation or joining.

Earlier works on thread-modular relational analysis rely on DATALOG rules to model interferences in the sense of Miné in combination with abstract interpretation applied to the Data-Flow Graph [19] or the Control-Flow Graph [31] (later extended to weak memory [32]), respectively. Botbol et al. [10] give a non-thread-modular analysis of multi-threaded programs with message-passing concurrency by encoding the program semantics as a symbolic transducer.

In all these approaches clusters of variables, if there are any, are predefined and not treated specially by the analysis. This is different in the thread-modular analysis proposed by Mukherjee et al. [42]. It propagates information from unlocks to locks. It is relational for the locals of each thread, and within *disjoint* subsets of globals, called *regions*. These regions must be determined beforehand and must satisfy *region-race freedom*. In contrast, the only extra a priori information required by our analysis, are the sets of (write-) protecting mutexes of globals – which can be computed during the analysis itself. The closest concept within our approach to a *region* is the set of globals jointly protected by mutexes. These sets may overlap – which the analysis explicitly exploits. Like ours, their proof of correctness refers to a thread-local semantics. Unlike ours, it is based on interleavings and thus overly detailed. The concrete semantics on which our analyses are based, is a collecting local trace semantics extending the semantics of Schwarz et al. [48] by additionally taking thread termination and joins into account. The analyses in [48], however, are non-relational. No refinement via further finite abstractions of local traces, such as thread *id*s is provided.

The thread *id* analysis perhaps most closely related to ours, is by Feret [20] who computes *id*s for agents in the π-calculus as abstractions of sequences of encountered create edges. Another line of analysis of concurrent programs deals with determining which critical events may happen in parallel (MHP) [1–4, 7, 17, 43, 59] to detect programming errors like, e.g., data races, or identifying opportunities for optimization. Mostly, MHP analyses are obtained as abstractions of a *global* trace semantics [18]. We apply related techniques for improving thread-modular analyses – but based on a *local* trace semantics. Like MHP analyses, we take thread creation and joining histories as well as sets of held mutexes into account. Additionally, we also consider crucial aspects of the modification history of globals and provide a general framework for further refinements.

In a sequential setting, splitting control locations according to some abstraction of reaching traces is a common technique for improving the precision of dataflow analyses [9, 27] or abstract interpretation [25, 34, 41, 47]. Control point splitting can be understood as an instance of the reduced cardinal power domain [12, 13, 22]. For the analysis of multi-threaded programs, Miné [39] applies the techniques of Mauborgne and Rival [34] to *single* threads, i.e., independently of the actions of all other threads. Our approach, on the other hand, may take arbitrary properties of *local* traces into account, and thus is more general.

11 Conclusion and Future Work

We have presented thread-modular relational analyses of global variables tailored to decomposable domains. In some cases, more precise results can be obtained by considering smaller clusters. For k-decomposable domains, however, we proved that the *optimal* result can already be obtained by considering clusters of size at most k. We have provided a framework to incorporate finite abstractions of local traces into the analysis. Here, we have applied this framework to take creation as well as joining of threads into account, but believe that it paves the way to seamlessly enhance the precision of thread-modular abstract interpretation. The evaluation of our analyses on benchmarks proposed in the literature indicates that our implementation is competitive both w.r.t. precision and efficiency. In future work, we would like to experiment with further abstractions of local traces, perhaps tailored to particular programming idioms, and also explore the potential of non-numerical 2-decomposable domains.

Acknowledgements. This work was supported by Deutsche Forschungsgemeinschaft (DFG) – 378803395/2428 ConVeY and the Estonian Centre of Excellence in IT (EXCITE), funded by the European Regional Development Fund.

References

1. Agarwal, S., Barik, R., Sarkar, V., Shyamasundar, R.K.: May-happen-in-parallel analysis of x10 programs. In: PPoPP '07, p. 183–193, ACM (2007), DOI: 10.1145/1229428.1229471
2. Albert, E., Flores-Montoya, A., Genaim, S.: Analysis of may-happen-in-parallel in concurrent objects. In: Giese, H., Rosu, G. (eds.) Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings, Lecture Notes in Computer Science, vol. 7273, pp. 35–51, Springer (2012), DOI: 10.1007/978-3-642-30793-5_3, URL https://doi.org/10.1007/978-3-642-30793-5_3
3. Albert, E., Genaim, S., Gordillo, P.: May-happen-in-parallel analysis for asynchronous programs with inter-procedural synchronization. In: Blazy, S., Jensen, T.P. (eds.) Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings, Lecture Notes in Computer Science, vol. 9291, pp. 72–89, Springer (2015), DOI: 10.1007/978-3-662-48288-9_5, URL https://doi.org/10.1007/978-3-662-48288-9_5
4. Albert, E., Genaim, S., Gordillo, P.: May-happen-in-parallel analysis with returned futures. In: D'Souza, D., Kumar, K.N. (eds.) Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings, Lecture Notes in Computer Science, vol. 10482, pp. 42–58, Springer (2017), DOI: 10.1007/978-3-319-68167-2_3, URL https://doi.org/10.1007/978-3-319-68167-2_3
5. Apinis, K., Seidl, H., Vojdani, V.: Side-effecting constraint systems: a swiss army knife for program analysis. In: APLAS '12, pp. 157–172, Springer (2012), DOI: 10.1007/978-3-642-35182-2_12
6. Arceri, V., Olliaro, M., Cortesi, A., Ferrara, P.: Relational string abstract domains. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings, Lecture Notes in Computer Science, vol. 13182, pp. 20–42, Springer (2022), DOI: 10.1007/978-3-030-94583-1_2, URL https://doi.org/10.1007/978-3-030-94583-1_2
7. Barik, R.: Efficient computation of may-happen-in-parallel information for concurrent Java programs. In: LCPC '06, vol. 4339 LNCS, pp. 152–169, Springer (2006), DOI: 10.1007/978-3-540-69330-7_11
8. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, p. 196–207, PLDI '03, Association for Computing Machinery, New York, NY, USA (2003), ISBN 1581136625, DOI: 10.1145/781131.781153, URL https://doi.org/10.1145/781131.781153
9. Bodík, R., Gupta, R., Soffa, M.L.: Refining data flow information using infeasible paths. SIGSOFT Softw. Eng. Notes **22**(6), 361–377 (Nov 1997), ISSN 0163-5948, DOI: 10.1145/267896.267921, URL https://doi.org/10.1145/267896.267921
10. Botbol, V., Chailloux, E., Gall, T.L.: Static analysis of communicating processes using symbolic transducers. In: Bouajjani, A., Monniaux, D. (eds.) Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings, Lecture Notes in Computer Science, vol. 10145, pp. 73–90, Springer (2017), DOI: 10.1007/978-3-319-52234-0_5, URL https://doi.org/10.1007/978-3-319-52234-0_5

11. Chen, L., Liu, J., Miné, A., Kapur, D., Wang, J.: An abstract domain to infer octagonal constraints with absolute value. In: Müller-Olm, M., Seidl, H. (eds.) Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8723, pp. 101–117, Springer (2014), DOI: 10.1007/978-3-319-10936-7_7, URL https://doi.org/10.1007/978-3-319-10936-7_7
12. 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), DOI: 10.4204/EPTCS.129.19, URL https://doi.org/10.4204/EPTCS.129.19
13. 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), DOI: 10.1145/567752.567778, URL https://doi.org/10.1145/567752.567778
14. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Why does astrée scale up? Form. Methods Syst. Des. **35**(3), 229–264 (dec 2009), ISSN 0925-9856, DOI: 10.1007/s10703-009-0089-6, URL https://doi.org/10.1007/s10703-009-0089-6
15. Cousot, P., Giacobazzi, R., Ranzato, F.: a^2i: Abstract^2 interpretation. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019), DOI: 10.1145/3290355, URL https://doi.org/10.1145/3290355
16. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978, pp. 84–96, ACM Press (1978), DOI: 10.1145/512760.512770, URL https://doi.org/10.1145/512760.512770
17. Di, P., Sui, Y., Ye, D., Xue, J.: Region-based may-happen-in-parallel analysis for C programs. In: ICPP, pp. 889–898, IEEE (2015), ISBN 978-1-4673-7587-0, DOI: 10.1109/ICPP.2015.98
18. Dwyer, M.B., Clarke, L.A.: Data flow analysis for verifying properties of concurrent programs. ACM SIGSOFT Software Engineering Notes **19**(5), 62–75 (dec 1994), DOI: 10.1145/195274.195295
19. Farzan, A., Kincaid, Z.: Verification of parameterized concurrent programs by modular reasoning about data and control. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, p. 297–308, POPL '12, Association for Computing Machinery, New York, NY, USA (2012), ISBN 9781450310833, DOI: 10.1145/2103656.2103693, URL https://doi.org/10.1145/2103656.2103693
20. Feret, J.: Abstract interpretation of mobile systems. J. Log. Algebraic Methods Program. **63**(1), 59–130 (2005), DOI: 10.1016/j.jlap.2004.01.005, URL https://doi.org/10.1016/j.jlap.2004.01.005
21. Fulara, J., Durnoga, K., Jakubczyk, K., Schubert, A.: Relational abstract domain of weighted hexagons. Electron. Notes Theor. Comput. Sci. **267**(1), 59–72 (2010), DOI: 10.1016/j.entcs.2010.09.006, URL https://doi.org/10.1016/j.entcs.2010.09.006
22. Giacobazzi, R., Ranzato, F.: The reduced relative power operation on abstract domains. Theor. Comput. Sci. **216**(1-2), 159–211 (1999), DOI: 10.

1016/S0304-3975(98)00194-7, URL https://doi.org/10.1016/S0304-3975(98)00194-7

23. Halbwachs, N., Merchat, D., Gonnord, L.: Some ways to reduce the space dimension in polyhedra computations. Formal Methods in System Design **29**(1), 79–95 (Jul 2006), ISSN 1572-8102, DOI: 10.1007/s10703-006-0013-2, URL https://doi.org/10.1007/s10703-006-0013-2
24. Halbwachs, N., Merchat, D., Parent-Vigouroux, C.: Cartesian factoring of polyhedra in linear relation analysis. In: Cousot, R. (ed.) Static Analysis, pp. 355–365, Springer Berlin Heidelberg, Berlin, Heidelberg (2003), ISBN 978-3-540-44898-3
25. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: Levi, G. (ed.) Static Analysis, pp. 200–214, Springer Berlin Heidelberg, Berlin, Heidelberg (1998), ISBN 978-3-540-49727-1
26. Heo, K., Oh, H., Yang, H.: Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In: Rival, X. (ed.) Static Analysis, pp. 237–256, Springer Berlin Heidelberg, Berlin, Heidelberg (2016), ISBN 978-3-662-53413-7
27. Holley, L.H., Rosen, B.K.: Qualified data flow problems. In: Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, p. 68–82, POPL '80, Association for Computing Machinery, New York, NY, USA (1980), ISBN 0897910117, DOI: 10.1145/567446.567454, URL https://doi.org/10.1145/567446.567454
28. Howe, J.M., King, A.: Logahedra: A new weakly relational domain. In: Liu, Z., Ravn, A.P. (eds.) Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings, Lecture Notes in Computer Science, vol. 5799, pp. 306–320, Springer (2009), DOI: 10.1007/978-3-642-04761-9_23, URL https://doi.org/10.1007/978-3-642-04761-9_23
29. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings, LNCS, vol. 5643, pp. 661–667, Springer (2009), DOI: 10.1007/978-3-642-02658-4_52, URL https://doi.org/10.1007/978-3-642-02658-4_52
30. Karr, M.: Affine relationships among variables of a program. Acta Informatica **6**, 133–151 (1976), DOI: 10.1007/BF00268497, URL https://doi.org/10.1007/BF00268497
31. Kusano, M., Wang, C.: Flow-sensitive composition of thread-modular abstract interpretation. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, p. 799–809, FSE 2016, Association for Computing Machinery, New York, NY, USA (2016), ISBN 9781450342186, DOI: 10.1145/2950290.2950291, URL https://doi.org/10.1145/2950290.2950291
32. Kusano, M., Wang, C.: Thread-modular static analysis for relaxed memory models. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, p. 337–348, ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA (2017), ISBN 9781450351058, DOI: 10.1145/3106237.3106243, URL https://doi.org/10.1145/3106237.3106243
33. Logozzo, F., Fähndrich, M.: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In: Proceedings of the 2008 ACM Symposium on Applied Computing, p. 184–188, SAC '08, Association for Computing Machinery, New York, NY, USA (2008), ISBN 9781595937537, DOI: 10.1145/1363686.1363736, URL https://doi.org/10.1145/1363686.1363736

34. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, M. (ed.) Programming Languages and Systems, pp. 5–20, Springer Berlin Heidelberg, Berlin, Heidelberg (2005), ISBN 978-3-540-31987-0
35. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) Programs as Data Objects, Second Symposium, PADO 2001, Aarhus, Denmark, May 21-23, 2001, Proceedings, LNCS, vol. 2053, pp. 155–172, Springer (2001), DOI: 10.1007/3-540-44978-7_10, URL https://doi.org/10.1007/3-540-44978-7_10
36. Miné, A.: The octagon abstract domain. In: WCRE' 01, p. 310, IEEE Computer Society (2001), DOI: 10.1109/WCRE.2001.957836
37. Miné, A.: The octagon abstract domain. Higher Order Symbol. Comput. **19**(1), 31–100 (mar 2006), ISSN 1388-3690, DOI: 10.1007/s10990-006-8609-1, URL https://doi.org/10.1007/s10990-006-8609-1
38. Miné, A.: Static analysis of run-time errors in embedded real-time parallel C programs. Logical Methods in Computer Science **8**(1), 1–63 (mar 2012), DOI: 10.2168/LMCS-8(1:26)2012
39. Miné, A.: Relational thread-modular static value analysis by abstract interpretation. In: VMCAI '14, vol. 8318 LNCS, pp. 39–58, Springer (2014), DOI: 10.1007/978-3-642-54013-4_3
40. Monat, R., Miné, A.: Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. In: VMCAI '17, vol. 10145 LNCS, pp. 386–404, Springer (2017), DOI: 10.1007/978-3-319-52234-0_21
41. Montagu, B., Jensen, T.: Trace-based control-flow analysis. In: PLDI '21, p. 482–496, ACM (2021), DOI: 10.1145/3453483.3454057, URL https://doi.org/10.1145/3453483.3454057
42. Mukherjee, S., Padon, O., Shoham, S., D'Souza, D., Rinetzky, N.: Thread-local semantics and its efficient sequential abstractions for race-free programs. In: SAS '17, vol. LNCS 10422, pp. 253–276, Springer (2017), DOI: 10.1007/978-3-319-66706-5_13
43. Naumovich, G., Avrunin, G.S., Clarke, L.A.: An efficient algorithm for computing mhp information for concurrent Java programs. In: ESEC/FSE '99, vol. 1687 LNCS, pp. 338–354, Springer (1999), DOI: 10.1007/3-540-48166-4_21
44. Oh, H., Heo, K., Lee, W., Lee, W., Park, D., Kang, J., Yi, K.: Global sparse analysis framework. ACM Trans. Program. Lang. Syst. **36**(3) (sep 2014), ISSN 0164-0925, DOI: 10.1145/2590811, URL https://doi.org/10.1145/2590811
45. Oh, H., Lee, W., Heo, K., Yang, H., Yi, K.: Selective x-sensitive analysis guided by impact pre-analysis. ACM Trans. Program. Lang. Syst. **38**(2) (Dec 2015), ISSN 0164-0925, DOI: 10.1145/2821504, URL https://doi.org/10.1145/2821504
46. Péron, M., Halbwachs, N.: An abstract domain extending difference-bound matrices with disequality constraints. In: Cook, B., Podelski, A. (eds.) Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings, Lecture Notes in Computer Science, vol. 4349, pp. 268–282, Springer (2007), DOI: 10.1007/978-3-540-69738-1_20, URL https://doi.org/10.1007/978-3-540-69738-1_20
47. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. **29**(5), 26–es (Aug 2007), ISSN 0164-0925, DOI: 10.1145/1275497.1275501, URL https://doi.org/10.1145/1275497.1275501
48. Schwarz, M., Saan, S., Seidl, H., Apinis, K., Erhard, J., Vojdani, V.: Improving thread-modular abstract interpretation. In: SAS '21, vol. 12913 LNCS, pp. 359–383, Springer (2021), DOI: 10.1007/978-3-030-88806-0_18

49. Schwarz, M., Saan, S., Seidl, H., Erhard, J., Vojdani, V.: Clustered relational thread-modular abstract interpretation with local traces. CoRR **abs/2301.06439** (2023), URL https://arxiv.org/abs/2301.06439
50. Schwarz, M., Saan, S., Seidl, H., Erhard, J., Vojdani, V.: Clustered Relational Thread-Modular Abstract Interpretation with Local Traces (Jan 2023), DOI: 10.5281/zenodo.7505428
51. Seidl, H., Vogler, R.: Three improvements to the top-down solver. Math. Struct. Comput. Sci. **31**(9), 1090–1134 (2021), DOI: 10.1017/S0960129521000499, URL https://doi.org/10.1017/S0960129521000499
52. Sharma, D., Sharma, S.: Thread-modular analysis of release-acquire concurrency. In: Dragoi, C., Mukherjee, S., Namjoshi, K.S. (eds.) Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings, LNCS, vol. 12913, pp. 384–404, Springer (2021), DOI: 10.1007/978-3-030-88806-0_19, URL https://doi.org/10.1007/978-3-030-88806-0_19
53. Simon, A., King, A., Howe, J.M.: Two variables per linear inequality as an abstract domain. In: Leuschel, M. (ed.) Logic Based Program Synthesis and Tranformation, 12th International Workshop, LOPSTR 2002, Madrid, Spain, September 17-20,2002, Revised Selected Papers, LNCS, vol. 2664, pp. 71–89, Springer (2002), DOI: 10.1007/3-540-45013-0_7, URL https://doi.org/10.1007/3-540-45013-0_7
54. Singh, G., Püschel, M., Vechev, M.: Fast polyhedra abstract domain. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, p. 46–59, POPL 2017, Association for Computing Machinery, New York, NY, USA (2017), ISBN 9781450346603, DOI: 10.1145/3009837.3009885
55. Singh, G., Püschel, M., Vechev, M.: Fast numerical program analysis with reinforcement learning. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification, pp. 211–229, Springer International Publishing, Cham (2018), ISBN 978-3-319-96145-3
56. Singh, G., Püschel, M., Vechev, M.: A practical construction for decomposing numerical abstract domains. Proc. ACM Program. Lang. **2**(POPL) (dec 2018), DOI: 10.1145/3158143, URL https://doi.org/10.1145/3158143
57. Suzanne, T., Miné, A.: Relational thread-modular abstract interpretation under relaxed memory models. In: APLAS '18, vol. LNCS 11275, pp. 109–128, Springer (dec 2018), DOI: 10.1007/978-3-030-02768-1_6
58. Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static Race Detection for Device Drivers: The Goblint Approach. In: ASE '16, pp. 391–402, ACM (2016), DOI: 10.1145/2970276.2970337
59. Zhou, Q., Li, L., Wang, L., Xue, J., Feng, X.: May-happen-in-parallel analysis with static vector clocks. In: CGO '18, pp. 228–240, ACM (2018), DOI: 10.1145/3168813

Adversarial Reachability for Program-level Security Analysis[⋆]

Soline Ducousso[1](✉), Sébastien Bardin[1](✉), and Marie-Laure Potet[2]

[1] Université Paris-Saclay, CEA, List, Saclay, France
soline.ducousso@cea.fr, sebastien.bardin@cea.fr
[2] Univ. Grenoble Alpes, VERIMAG, Grenoble, France
marie-laure.potet@univ-grenoble-alpes.fr

Abstract. Many program analysis tools and techniques have been developed to assess program vulnerability. Yet, they are based on the standard concept of reachability and represent an attacker able to craft smart *legitimate* input, while in practice attackers can be much more powerful, using for instance micro-architectural exploits or fault injection methods. We introduce *adversarial reachability*, a framework allowing to reason about such *advanced attackers* and check whether a system is vulnerable or immune to a particular attacker. As equipping the attacker with new capacities significantly increases the state space of the program under analysis, we present a new symbolic exploration algorithm, namely *adversarial symbolic execution*, injecting faults in a *forkless* manner to prevent path explosion, together with optimizations dedicated to reduce the number of injections to consider while keeping the same attacker power. Experiments on representative benchmarks from fault injection show that our method significantly reduces the number of adversarial paths to explore, allowing to scale up to 10 faults where prior work timeout for 3 faults. In addition, we analyze the well-tested WooKey bootloader, and demonstrate the ability of our analysis to find attacks and evaluate countermeasures in real-life security scenarios. We were especially able to find an attack not mentioned in a previous patch.

Keywords: Program analysis · Attacker model · Fault injection · Symbolic execution

1 Introduction

Context. Major works have delved into program analysis over the last decades, leveraging techniques such as symbolic execution [24,53,18], static analysis [43], abstract interpretation [30] or bounded model checking [29], to hunt for software vulnerabilities and bugs in programs, or to prove their absence [35,60], leading to industrial adoption in some leading companies [18,43,6,60,66]. As bugs are an attack entry point, removing them is a first step towards better software security.

⋆ Partially supported by grants ANR TAVA, PEPR Secureval and Carnot Flexsecurity.

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 59–89, 2023.
https://doi.org/10.1007/978-3-031-30044-8_3

Problem. Yet, stepping back from these successes, it appears that all these methods consider a rather weak threat model, where the attacker can only craft smart "inputs of death" through legitimate input sources of the program, exploiting corner cases in the code itself. Tools only looking for bugs and software vulnerabilities may deem a program secure while the bar remains quite low for an *advanced attacker*, able for example to take advantage of attack vectors such as (physical) hardware fault injections [58], micro-architectural attacks [61,70], software-based hardware attacks [86,55,69] like Rowhammer [70], or any combination of vectors [63]. While previously limited to high-security devices and systems such as smart cards and cryptography modules [16,13], these fault-based attacks can now target a wider spectrum of systems, such as bootloaders [57], firmware update modules [19], security enclaves [69], etc. The reasoning behind automated software-implemented fault injection also applies to Man-At-The-End attacks [3] and is similar to the (manual) reasoning performed in control-flow integrity to evaluate countermeasures [1,21].

Goal & Challenges. *Our goal is to devise a technique to automatically and efficiently reason about the impact of an advanced attacker onto program security properties*, where the standard reachability framework only supports an attacker crafting smart legitimate inputs. The first challenge is to provide a formal framework to study what an advanced attacker can do to attack a program. Interestingly, while such frameworks are routinely used in cryptographic protocol verification [26,7], none has been studied for program-level analysis. The second challenge is to design an efficient algorithm to assess the vulnerability of a program to a given attacker model, while adding capabilities to the attacker naturally gives rise to a significant path explosion – especially in the case of multiple fault analysis.

The rare prior works in the field, mostly focused on encompassing physical fault injections for high-security devices, rely mostly on *mutant generation* [28,79,49,25,50] or *forking analysis* [76,15,20,63], yielding scalability issues. Moreover, most of them are limited to a few predefined fault models and do not propose any formalization of the underlying problem.

Proposal. We propose *adversarial reachability*, a formalism extending standard reachability to reason about a program execution in the presence of an advanced attacker, and we build a new algorithm based on symbolic techniques, named *adversarial symbolic execution*, to address the adversarial reachability problem from the bug finding point of view (bounded verification). Our algorithm prevents path explosion thanks to a new *forkless* encoding of faults. We show it is correct and k-complete with respect to adversarial reachability. To improve the performance further, we design two new optimizations to reduce the number of injected faults: Early Detection of fault Saturation and Injection On Demand.

Contributions. As a summary, we claim the following novelties:

– We formalize the adversarial reachability problem (Section 4), extending standard reachability to take into account an advanced attacker, together with the associated correctness and completeness definitions;

– We describe a new symbolic exploration method (Section 5), adversarial symbolic execution, to answer adversarial reachability, featuring a novel forkless fault encoding to prevent path explosion and two optimization strategies to reduce fault injection. We establish their correctness and completeness;
– We propose an implementation of our techniques for binary-level analysis (Section 6), on top of the BINSEC framework [38]. We systematically evaluate its performances against prior work (Section 7), using a standard SWiFI benchmark from physical fault attacks and smart cards. Experiments show a very significant performance gain against prior approaches, for example up to x10 and x215 times on average for 1 and 2 faults respectively – with a similar reduction in the number of adversarial paths. Moreover, our approach scales up to 10 faults whereas the state-of-the-art starts to timeout for 3 faults ;
– We finally perform a security analysis of the WooKey bootloader [1] (Section 8), a very well tested real-life security-focused program. We were able to find known attacks and evaluate the adequacy of some of the countermeasures. Especially, we found an attack not taken into account in a recently proposed patch [63], and proposed a new patch to the developers.

This work is a first step in designing efficient program analysis techniques able to take into account advanced attackers. The approach is generic enough to accommodate many common fault models, including the bit flip from RowHammer, test inversion or arbitrary data modification; still, instruction skips or modifications are currently out of reach. Also, while we investigate the bug finding side of the problem (underapproximation), the verification side (overapproximation) is interesting as well. These are exciting directions for future research.

Our dataset and benchmark infrastructure are made available through artifact[2] *for reproducibility purpose, and the code is open-sourced*[3].

2 Motivation

We start by motivating the need for adversarial reachability, first with a description of several realistic attack scenarios on software involving advanced attackers (Section 2.1), second with a small example showing the need for dedicated analysis (Section 2.2).

2.1 Fault Injection across Security Fields

We describe hereafter several real software-level security scenarios where the attacker goes beyond crafting legitimate input to abuse the system under at-

[1] WooKey [14,89] is a secure USB mass storage device developed by the French National Security Agency, and has recently served as a recent challenge among French security evaluators.

[2] DOI: 10.5281/zenodo.7507112
`https://zenodo.org/record/7507112#.Y7cLsKfMJhE`

[3] `https://github.com/binsec/binsec-ase`

tack. Interestingly, while these scenarios were historically focused on hardware-hardened high-security systems (such as smart cards) and associated with complex physical attack means, many recent scenarios do involve software-only attacks on standard systems, with targets encompassing cryptographic libraries, bootloaders, firmware updaters, security enclaves, etc.

Hardware Fault Injection Attacks [58] cause erroneous computations by disturbing signal propagation in the chip with physical means such as electromagnetic pulses [39], laser beams [85,4], or power [19] and clock glitches. The associated fault models include bit-, byte- or word- set and reset, bit-flips, instructions corruption and instruction skips. State-of-the-art attacks involve multiple fault injections [59], as expected by the high level of attack potential in Common Criteria vulnerability analysis.

Software-implemented Hardware Attacks push the hardware into unstable states using software controlled mechanisms, like delays in memory buses inducing bit-flips in data fetched from memory [55] or CPU voltage and frequency manipulations yielding bit-flips in the processor [86,69]. The notorious *Rowhammer* attack [70] abuses memory accesses to induce bit-flips in flash memory.

Micro-architectural Attacks use micro-architectural behaviors in unexpected ways. For example: Spectre (version v1) [62] exploits branch predictors in speculative executions, which can be seen as a test inversion followed by a rollback; Load Value Injection [87] injects arbitrary data into transient execution; race attacks [54] corrupt data of other running processes and can be seen as arbitrary data faults.

Man-At-The-End Attacks considers an attacker having full observability and control over a software code and its execution [3], with the goal to steal sensitive data or code (reverse engineering attacks). The associated attacker model is hence very powerful, with capabilities such as halting and modifying data and code at any point of the execution.

CFI Reasoning In order to assess the power of Control-Flow Integrity (CFI) mechanisms, researchers [1,21] define hypothetical attackers by their capabilities, such as "write anything anywhere" or "write anything somewhere", and manually prove that their countermeasure is indeed able to thwart such an opponent. While not *per se* an applicative security scenario, the techniques developed in this paper could help automate such essential reasoning.

2.2 Motivating Example

The motivating example in Figure 1 is a simple unrolled program inspired by the VerifyPIN benchmark [42], from the domain of hardware fault injection and smart cards. The user PIN digits $u1$ to $u4$ are checked against the reference digits $ref1$ to $ref4$, using the accumulator res. The attacker seeks to be authenticated (validate the assert l.16) without knowing the right digits (l.14).

```
 1 bool g_authenticated;
 2 int u1, u2, u3, u4, ref1, ref2, ref3, ref4;
 3
 4 void verifyPIN() {
 5     int res = 1;
 6     res = res * (u1 == ref1);
 7     res = res * (u2 == ref2);
 8     res = res * (u3 == ref3);
 9     res = res * (u4 == ref4);
10     g_authenticated = res;
11 }
12
13 void main(int argc, char const *argv[]) {
14     assert(u1!=ref1 || u2!=ref2 || u3!=ref3 || u4!=ref4);
15     verifyPIN();
16     assert(g_authenticated == true); /* Security oracle */
17 }
```

Fig. 1: Motivating example, inspired by VerifyPIN [42]

Here, the attacker indeed cannot succeed by only crafting legitimate inputs. However, an advanced attacker can leverage more powerful attack vectors to inject faults into the program in order to succeed. For instance, corrupting *g_authenticated* to *true* at l.10 achieves the attacker goal. It could be obtained for example through a physical- or Rowhammer- attack.

Program Analysis As expected, standard symbolic execution tools such as Klee [22], angr [84] or BINSEC [38] do not report any violation here, as they consider the simplest possible attacker. We can try to use SWiFI techniques [76,15,20,63] (detailed in Section 3.1) from high-security system evaluation. Yet, the standard *forking* approach does not scale with multiple faults: here, 166 paths are explored in 0.6 seconds for 1 fault, 2994 paths in 11 seconds for 2 faults, and it keeps on adding a factor x10 in explored paths and analysis time for each extra fault, until the analysis timeouts (12 hours) above 4 faults. On the contrary, our *forkless* algorithm presented in Section 5 simulates fault injection without creating new paths and, in this example, shows a constant runtime as the number of faults increases from 1 to 10 – we explore 9 paths in 0.2 seconds in all cases.

3 Background

We provide in this section background information on software-implemented fault injection, standard reachability and symbolic execution.

3.1 Software-implemented Fault Injection (SWiFI)

SWiFI tools [28,76,79,15,49,25,20,50,63,68] have been developed in the community of high-secure systems to ease hardware fault injection campaigns, which are time consuming and require special equipment. SWiFI evaluates a program with the transformations induced by the effects of hardware faults, in order to find interesting attack paths. We distinguish two main SWiFI techniques.

First, the *Mutant generation* approach [28,79,49,25,50] consists in analyzing slightly modified versions of the program (named mutants), each of them embedding a different faulty instruction. Each mutant is then analyzed on its own. The main limitation of mutant generation is the explosion of mutants, in particular for multiple faults. Also, as the different mutants differ only slightly, analyzing each of them separately wastes lots of time repeating similar reasoning.

```
x := y + z
```

(a) Original statement

```
if (fault_here)
   then x := fault_value
   else x := y + z
```

(b) Forking transformation

Fig. 2: Forking code transformation in pseudo-code

Second, the *forking approach* [76,15,20,63] consists in instrumenting the analysis (or the code, via instrumentation) to add all possible faults as forking points (branches) controlled by boolean values indicating whether a particular fault will be taken or not, plus constraints on the maximal number of faults allowed. A forking data fault is illustrated in Figure 2. A standard program analysis technique is then launched – typically symbolic execution or bounded model checking. Compared with mutant generation, this method allows sharing the analysis between the different possible faults. Still, the number of paths explodes with the number of possible faults (forking points).

Scalability Issues These two approaches yield an explosion of the whole search space w.r.t. the number of fault injection points in the program: the mutant approach leads to consider up to C_k^n (k among n)[4] mutants for a program under analysis with n possible fault locations and k faults, while the forking approach yields up to C_k^n paths to analyzed for a single original program path with n possible fault locations and k faults.

In the following, we will consider the forking approach as the baseline – please keep in mind that the mutant approach scales worse.

Fault Models Supported fault models vary for each tool, but they are usually adapted from hardware fault models [47,82]. The most common fault models are (1) *data faults* such as arbitrary data modifications, set and reset of bytes, words or variables, bit-flips; and (2) *instruction corruptions* such as instruction skips

[4] Remind that $C_k^n = \binom{k}{n} = \frac{n!}{k!(n-k)!}$

and test inversions. Most tools are limited to one (sometimes two) hard-coded fault models. Only few SWiFI tools can handle multiple faults [88,76,63,68] – still with scalability issues.

3.2 Standard Reachability Formalization

Considering a program P, we denote S the set of all possible states of P. A state is composed of the code memory, the data memory (i.e. the stack and heap), the state of registers and the location of the next instruction to execute. The set of input states of a program P is noted $S_0 \subset S$. The set of transitions (or instructions) of the program is denoted T. The execution of an instruction t is represented by a one-step transition relation $\rightarrow_t \in S \times S$. We denote $s \rightarrow s'$ when $s \rightarrow_t s'$ for some $t \in T$. We extend the transition relation over any finite path $\pi \in T^*$ through composition. The transitive reflexive closure of $\rightarrow$ is noted $\rightarrow^*$. Finally, we use $S \rightarrow s'$ as a shortcut for $\exists s \in S. s \rightarrow s'$, and $\rightarrow_{\leq k}$ for reachability in at most k steps.

We consider in the rest of the paper the case of *location reachability*: given a location l (instruction or code address) of the program under analysis, the question is whether we can reach any state s at location l. More formally, L is the finite set of locations of P, and we consider a mapping $loc : S \mapsto L$ from states to locations. For example, loc may return the program counter value. We write $S \rightarrow^* l$ as a shortcut for $\exists s' \in S. S \rightarrow^* s' \wedge loc(s') = l$.

Definition 1 (Standard reachability). *A location l is reachable in a program P if $S_0 \rightarrow^* l$.*

We now define correctness and completeness for a program analyzer.

Definition 2 (Correctness, completeness). *Let $\mathcal{V} : (P, l) \mapsto \{1, 0\}$ be a verifier taking as input a program P and a target location l.*

- *$\mathcal{V}$ is correct when for all P, l, if $\mathcal{V}(P, l) = 1$ then l is reachable in P ;*
- *$\mathcal{V}$ is complete when for all P, l, if l is reachable then $\mathcal{V}(P, l) = 1$;*
- *if $\mathcal{V}$ also takes an integer bound n as input, $\mathcal{V}$ is k-complete when for all bound n and P,l, if l is reachable in at most n steps then $\mathcal{V}(P, l, n) = 1$.*

We want to stress out that while location reachability can be seen as a basic case, we consider it sufficient here for two reasons: first, it keeps the formalism light while still straightforward to generalize to stronger reachability properties (e.g., local predicates of the form (l, φ), sets of finite traces, etc.); second, it is already rather powerful on its own, as we can still instrument the code to reduce some stronger forms of reachability to it (e.g., adding local assertions or monitors).

3.3 Symbolic Execution

Symbolic execution (SE) [52,83,23,24] is a symbolic exploration technique for standard reachability. Algorithm 1 gives a high-level view of a typical SE al-

Algorithm 1: Standard symbolic execution algorithm, taken from [48]

```
Input: a program $P$, a bound $k$, a target location $l$
Output: Boolean value indicating whether $l$ can be reached within $k$ steps.
1 for path $\pi$ in GetPaths($k$) do
2     if $\pi$ reaches $l$ then
3         $\Phi$ := GetPredicate($\pi$)
4         if $\Phi$ is satisfiable then
5             return true
6         end
7     end
8 end
9 return false
```

gorithm, adapted for location reachability[5]. The analysis follows each possible path π of a program up to a depth bound k. If π reaches the target, then we check whether π is indeed feasible by computing its *path predicate* Φ – a logical formula representing the path constraints over the input variables along π, and sending it to a SMT solver [12], that will try to answer whether the formula is satisfiable or not, and provide a model for free variables (e.g. inputs) if it is (omitted here for simplicity). SE is *correct* for location reachability, and even *k-complete* if we assume a perfect encoding of path predicates.

Algorithm 2: Assignment evaluation in SE

```
Input: path predicate $\Phi$, assignment instruction $x := expr$
Output: Updated $\Phi$
1 Function eval_assign($\Phi$, $x$, $expr$) is
2     return $\Phi \wedge (x \triangleq expr)$
3 end
```

In this paper, we will focus on the evaluation of assignments and conditional jumps for SE, detailed in Algorithms 2 and 3 respectively, as this is where our adversarial symbolic execution will mainly differ from the standard one. It requires going slightly deeper into details. In practice, the program paths are explored incrementally. A worklist WL records all pending paths together with their associated path predicate and their next instruction to explore. On conditional branches, the symbolic path is split in two (one for each branch, updating the path constraint accordingly), and each new prefix is added to the worklist (Al-

[5] More complex properties can be verified with the same principles, such as local predicate reachability, trace properties or hyper-properties [36].

Algorithm 3: Conditional jump evaluation in SE

```
Input: path predicate $\Phi$, conditional jump instruction $if\ cdt\ then\ l_t\ else\ l_e$
Data: a worklist $WL$ containing the pending path prefixes to explore – list of
      pairs (path predicate, next location)
Output: $WL$ updated in place

Function eval_conditional_jump($\Phi$, $cdt$, $l_t$, $l_e$) is
    if $\Phi \wedge cdt$ is satisfiable then
        Add $(\Phi \wedge cdt, l_t)$ to $WL$
    end
    if $\Phi \wedge (\neg cdt)$ is satisfiable then
        Add $(\Phi \wedge \neg cdt, l_e)$ to $WL$
    end
end
```

gorithm 3). Assignments are dealt with straightforwardly, simply adding a new logical variable definition to the path predicate [6] (notation: $x \triangleq y$).

4 Adversarial Reachability

In this section, we detail the advanced attacker model we consider and define the adversarial reachability problem. Especially, *advanced attackers can do more than carefully crafting legitimate inputs to trigger vulnerabilities in a software.* They can use a wide variety of attack vectors (e.g. hardware fault injection attacks, software-implemented hardware attacks, micro-architectural attacks, software attacks, etc), in any combination, and multiple times. We suppose attack vectors prerequisites have been met, and only consider the impact of the faults on the program under attack.

Our *attacker model* has three components: (1) a set of attacker actions, equivalent to fault models; (2) the maximum number of actions the attacker can perform; and (3) a goal, expressed here as a location reachability query.

Formally, given a program P with set of states S, set of transitions T and set of locations L, we extend the transition model described in Section 3.2 to include an adversarial transition $\rightsquigarrow_A \in S \times S$ related to an attacker A, i.e. $T_A = T \cup \rightsquigarrow_A$. To specify practical fault models, restrictions are applied onto $\rightsquigarrow_A$, limiting what part of the state can be modified and how. For instance, when considering arbitrary data faults, only the data memory and the register values can be modified. Then, the transition relation of P under attacker A is denoted as $\mapsto_A = \rightarrow \cup \rightsquigarrow_A = (\cup_{t \in T} t) \cup \rightsquigarrow_A$. We extend the notations from Section 3.2 to the relation $\mapsto_A$. Especially, $S \mapsto_A^* s'$ means $\exists s \in S. s \mapsto_A^* s'$, the adversarial transition relation up to k is denoted $\mapsto_{A,\leq k}$.

[6] Actually, a symbolic state usually comprises the path predicate itself plus a mapping from program variable names to logical variable names, and assignments involve both creating new logical names and updating the mapping. We abstract away from these details.

Still, we need to take into account the maximum number of faults the attacker can perform along an execution. Given a path π over T_A^*, π is said to be *legit* if it does not contain $\rightsquigarrow_A$, and *faulty* otherwise. The number of occurrences of transition $\rightsquigarrow_A$ in π is its *number of faults.* Given a bound m_A on the fault capability of A, we define $\mapsto^*_{(A,m_A)}$ by limiting the adversarial reachability relation to paths π with less than m_A faults. We consider m_A to be $+\infty$ in case the attacker has no such limitation. For the sake of simplicity, in the following, we will consider m_A as an implicit parameter of A, and simply write $\mapsto^*_A$ instead of $\mapsto^*_{(A,m_A)}$.

Definition 3 (Adversarial reachability). *Given an attacker A with a m_A faults budget and a program P, a location $l \in L$ is adversarially reachable if $S_0 \mapsto^*_A s' \wedge loc(s') = l$ for some $s' \in S$.*

In the following, adversarial reachability of location l from a set of states S_0 will be denoted $S_0 \mapsto^*_A l$.

Proposition 1. *Standard reachability implies adversarial reachability. The converse does not hold.*

Proof. Standard reachability can be viewed as adversarial reachability with an attacker able to perform 0 faults.

We redefine what it means for an analysis answering adversarial reachability to be correct, complete and k-complete.

Definition 4. *Let $\mathcal{V}_A : (P, A, l) \mapsto \{1, 0\}$ be a verifier taking as input a program P, an attacker A with m_A fault budget and a target location l.*

- *$\mathcal{V}_A$ is correct given A when for all P, l, if $\mathcal{V}_A(P, A, l) = 1$ then l is adversarially reachable in P for attacker A;*
- *$\mathcal{V}_A$ is complete given A when for all P, l, if l is adversarially reachable for attacker A then $\mathcal{V}_A(P, A, l) = 1$;*
- *if $\mathcal{V}_A$ also takes an integer bound n as input, $\mathcal{V}_A$ is k-complete given A when for all integer n and P,l, if l is adversarially reachable in at most n steps then $\mathcal{V}_A(P, A, l, n) = 1$.*

5 Forkless Adversarial Symbolic Execution (FASE)

In this section, we present our forkless algorithm for adversarial reachability. The analysis aims to find inputs and a fault sequence compatible with the considered attacker model and reaching the target location. Our primary goal is to deal with the potential path explosion induced by possible faults. Our design guiding principles are the following:

- First, prevent path explosion as much as possible with a forkless fault encoding. Yet, this forkless encoding leads to logical formulas potentially more complex and harder to solve in practice;
- Second, reduce as much as possible the complexity of the created formulas, by avoiding the undue introduction of extra-faults along a path.

5.1 Modelling Faults via Forkless Encoding

The forkless encoding aims to address the path explosion induced by the forking treatment of fault injection in prior works. It is designed mainly for data faults and consists of wrapping arithmetically an assignment right-hand side, as shown in Figure 3 for an arbitrary data fault. The activation of this fault location is determined by the symbolic Boolean value $fault\_here$, and the corrupted value of x is the fresh variable $fault\_value$.

The point is to embed the fault injection as an expression inside the logical formula, without any explicit path forking at the analysis top-level, in order to let the analyzer reason about both legit executions and faulty executions at the same time – this is akin to path merging in some ways, except that we do it only for the treatment of fault injection (we could also see the approach as avoiding undue path splits).

Multiple forkless arbitrary data encodings are possible. We chose to use the *ite* expression operator, an inlined form of if-then-else at the expression level. We also tried encodings inspired from branchless programming idioms (e.g.: $(b) \cdot x + (1-b) \cdot y$. for $ite(b, x, y)$ with b a Boolean value) – in our experiments they worked as well as the *ite* operator. Other data fault models are supported, such as set, reset, bit-flips, etc. Test inversion is also supported by applying faults to the condition of conditional jumps. Table 1 illustrates various forkless encodings. Note that the forkless encoding is not designed for instruction corruptions or instruction skips, as these modifications either yield permanent code modification or span several instructions.

```
x := expr
```

(a) Original statement

```
x:=ite fault_here? fault_value:expr
```

(b) Forkless transformation for arbitrary data fault

Fig. 3: Forkless injection technique

Table 1: Forkless encodings for various fault models

Fault model	original instruction	Forkless encoding
Arbitrary data	$x := expr$	$x := ite\ fault\_here\ ?\ fault\_value\ :\ expr$
Variable reset	$x := expr$	$x := ite\ fault\_here\ ?\ 0x00000000\ :\ expr$
Variable set	$x := expr$	$x := ite\ fault\_here\ ?\ 0xffffffff :\ expr$
Bit-flip	$x := expr$	$x := ite\ fault\_here\ ?$ $(expr\ xor\ 1 << fault\_value) :\ expr$
Test inversion	$if\ cdt\ then\ goto\ 1$ $else\ goto\ 2$	$if\ (ite\ fault\_here\ ?\ !cdt\ :\ cdt)$ $then\ goto\ 1\ else\ goto\ 2$

Trade-off. While these sorts of encoding indeed allow a significant path reduction compared to forking approaches, the corresponding path predicates are more complicated than standard path predicates, as they involve lots of extra-symbolic variables for deciding whether the faults occur and for emulating their effect. We show later in this section how to reduce these extra-variables.

5.2 Building Adversarial Path Predicates

Adversarial symbolic execution requires modifications to Algorithms 2 and 3, as illustrated in Algorithms 4 and 5 respectively.

Algorithm 4: Forkless assignment evaluation

```
Input: path predicate $\Phi$, assignment instruction $x := expr$, current number of
       faults $nb_f$
Output: Updated $\Phi$
1 Function eval_assign($\Phi$, $x$, $expr$) is
2     $\Phi'$, $expr'$, $nb_f$ := FaultEncoding($\Phi$, $expr$, $nb_f$)
3     return $\Phi' \wedge (x \triangleq expr')$
4 end
```

Algorithm 5: Forkless conditional jump evaluation

```
Input: path predicate $\Phi$, conditional jump instruction $if\ cdt\ l_t\ else\ l_e$
Data: fault counter $nb_f$, maximal number of faults $max_f$, worklist $WL$
Output: $WL$ updated in place
1 Function eval_conditional_jump($\Phi$, $cdt$, $l_t$, $l_e$) is
2     if $\Phi \wedge cdt \wedge (nb_f \leq max_f)$ is satisfiable then
3         Add ($\Phi \wedge cdt$, $l_t$) to $WL$
4     end
      /* Idem for else branch ($\neg cdt$)                */
5 end
```

The assign evaluation process embeds a wrapper encoding the fault in a forkless manner. Note that *FaultEncoding* involves the declaration of fresh symbolic variables for fault decisions and fault effects – hence the update of the path predicate Φ. Also, the fault counter nb_f is updated, and a new potentially faulted expression $expr'$ is computed.

Note that checking if the fault counter nb_f does not exceed the maximal number of faults max_f can be performed at different places. We found the best trade-off is to augment the conditional jump queries to check if we could explore each branch without exceeding max_f (see Algorithm 5), as checking at the end of a path often involves exploring many unfeasible faulty paths.

We refer to this set of modifications as Forkless Adversarial Symbolic Execution (FASE).

5.3 Algorithm Properties

We now consider the properties of the FASE algorithm.

Proposition 2. *The FASE algorithm is correct and k-complete for adversarial reachability.*

Sketch of proof. If our algorithm finds an adversarial path reaching the target location l, by providing specific input values and a fault sequence, then an attacker executing the program with the provided inputs and performing the proposed faults will reach its goal. Our algorithm is based on symbolic execution with bounded path depth and explores all possible attack paths according to the considered attacker model, hence its k-completeness for adversarial reachability.

Tightness of FASE. Consider a single path with no branching instruction and an assert statement to be checked at the end, together with f possible fault locations and a maximum of m faults. Then the forking SE yields up to C_m^f paths to analyze, and as many queries to send to the solver. In the same scenario, FASE will analyze only the original path, and *send a single query to the solver.*

Still, the Forkless encoding increases query complexity, as shown in Section 7. We present in the remainder of this section two mitigation techniques.

5.4 Optimization via Early Detection of Fault Saturation (FASE-EDS)

Algorithm 6: FASE-EDS conditional jump evaluation

Input: path predicate Φ, conditional jump instruction $if\ cdt\ then\ l_t\ else\ \ l_e$
Data: fault counter nb_f, maximal number of faults max_f, worklist WL
Output: WL updated in place

```
1 Function eval_conditional_jump_EDS($\Phi$, $cdt$, $l_t$, $l_e$) is
2     if $\Phi \wedge cdt \wedge (nb_f < max_f)$ is satisfiable then
3         Add ($\Phi \wedge cdt$, $l_t$) to $WL$
4     else if $\Phi \wedge cdt \wedge (nb_f == max_f)$ is satisfiable then
5         Stop injection in this path
6         Add ($\Phi \wedge cdt$, $l_t$) to $WL$
7     end
      /* Idem for else branch ($\neg cdt$)                          */
8 end
```

The first angle we explore to minimize query complexity is to reduce the number of injection points by *stopping the injection process as soon as possible.* Indeed, fewer injection points mean fewer extra symbolic variables and in general smaller and simpler queries for the SMT solver. We call this optimization *Early Detection of fault Saturation*, and write FASE-EDS when it is activated.

Its difference compared to FASE is in handling conditional jumps, illustrated in Algorithm 6. Instead of checking whether a branch can be explored without exceeding the maximum number of faults, we double the check: (1) first we check whether the branch can be explored with strictly fewer faults than allowed. If the query is satisfiable, the analysis continues down that branch as usual; (2) if not satisfiable, we check whether the branch is feasible with exactly the maximal number of faults allowed. If not, the branch is infeasible and we stop as usual. Yet, if it is feasible, then we know that we have spent all allowed faults. We can thus continue the exploration *without injecting any new fault* in the corresponding search sub-tree, leading to simpler subsequent queries.

Proposition 3. *FASE-EDS is correct and k-complete for the adversarial reachability problem.*

Proof. FASE-EDS remains correct as it does not modify the path predicate computation, and it remains k-complete as it only prunes fault injections that are actually infeasible – and would have been proven so by the solver, later in the solving process.

5.5 Optimization via Injection on Demand (FASE-IOD)

The second angle explored to reduce query complexity through the reduction of injection points is *to inject faults on demand*, only when they are truly needed. We call this optimization *Injection On Demand*, and write FASE-IOD when it is activated.

To inject faults on demand, we now build *two* path predicates along a path: the working path predicate Φ based on which solver queries are built (where we try to minimize fault injection), and the normal adversarial path predicate Φ_F computed in previous sections (encompassing all the faults seen so far).

Algorithm 7: FASE-IOD assignment evaluation

```
Input: path predicate $\Phi$, faulted path predicate $\Phi_F$, assignment instruction
       $x := expr$, current number of faults (in $\Phi_F$) $nb_f$
Output: Updated $\Phi$, $\Phi_F$
1 Function eval_assign_IOD($\Phi$, $\Phi_F$, $cdt$, $x$, $expr$) is
2     $\Phi'_F$, $expr'$, $nb_f$ := FaultEncoding($\Phi_F$, $expr$, $nb_f$)
3     return ($\Phi \wedge (x \triangleq expr)$, $\Phi'_F \wedge (x \triangleq expr')$)
4 end
```

Algorithm 8: FASE-IOD conditional jump evaluation

```
Input: path predicate $\Phi$, conditional jump instruction $if\ cdt\ then\ l_t\ else\ \ l_e$
Data: fault counter $nb_f$, maximal number of faults $max_f$, under
      approximation counter $under\_counter$, worklist $WL$
Output: $WL$ updated in place
1  Function eval_conditional_jump_IOD($\Phi$, $\Phi_F$, $cdt$, $l_t$, $l_e$) is
2      if $\Phi \wedge cdt \wedge (nb_f \leq max_f)$ is satisfiable then
3          Add $(\Phi \wedge cdt, \Phi_F \wedge cdt, l_t)$ to $WL$
4      else if $under\_counter \leq max_f$ then
5          if $\Phi_F \wedge cdt \wedge (nb_f \leq max_f)$ is satisfiable then
6              $\Phi := \Phi_F$
7              $under\_counter := under\_counter + 1$
8              Add $(\Phi \wedge cdt, \Phi_F \wedge cdt, l_t)$ to $WL$
9          end
10     end
       /* Idem for else branch ($\neg cdt$)                                    */
11 end
```

Algorithms are updated accordingly. Especially, assignment evaluation is duplicated as shown in Algorithm 7: The normal symbolic assignment, with the original right-hand-side expression *expr*, is added to Φ, while Φ_F is updated with the fault encoding of the assignment, $expr'$.

The on-demand reasoning takes place in the conditional jump instruction process detailed in Algorithm 8. The basic idea is to first check branch feasibility with the simpler path predicate Φ, encompassing the least number of faults. We continue this way as long as we can, meaning we rely on standard reachability as much as we can.

When encountering a branch infeasible with Φ, we then check whether this branch is feasible with all the possible faults seen so far, i.e. using Φ_F. If no that is a stop, otherwise we know that Φ does not encompass enough faults to go further. We then replace Φ by Φ_F (called a *switch*) at this stage, and thus continue with strictly more faults. Note that this is straightforward as Φ_F and Φ only differ on fault injections. Then again, the new Φ will not accumulate any fault (until a new switch) while Φ_F continues accumulating all possible faults.

As a bonus, the number of path predicate switches gives us an under-approximation $under\_counter$ of the number of faults already needed in the path under analysis. We use it to stop the injection early, when at least max_f faults have been used.

Proposition 4. *FASE-IOD is correct and k-complete for the adversarial reachability problem.*

Proof. FASE-IOD explores the same feasible paths as FASE, hence preserving its properties.

5.6 Optimizations Combination

Algorithm 9: FASE-IOD and FASE-EDS combination, conditional jump evaluation

```
Input: path predicate $\Phi$, faulty path predicate $\Phi_F$, conditional jump
       instruction $if\ cdt\ then\ l_t\ else\ \ l_e$
Data: fault counter $nb_f$, maximal number of faults $max_f$, under
      approximation counter $under\_counter$, worklist $WL$
Output: $WL$ updated in place

1  Function eval_conditional_jump_EDS_IOD($\Phi$, $\Phi_F$, $cdt$, $l_t$, $l_e$) is
2      if $\Phi \wedge cdt \wedge (nb_f \leq max_f)$ is satisfiable then
3          Add $(\Phi \wedge cdt, \Phi_F \wedge cdt, l_t)$ to $WL$
4      else if $under\_counter \leq max_f$ then
5          if $\Phi_F \wedge cdt \wedge (nb_f < max_f)$ is satisfiable then
6              $\Phi := \Phi_F$
7              $under\_counter := under\_counter + 1$
8              Add $(\Phi \wedge cdt, \Phi_F \wedge cdt, l_t)$ to $WL$
9          else if $\Phi_F \wedge cdt \wedge (nb_f == max_f)$ is satisfiable then
10             $\Phi := \Phi_F$
11             Stop $\Phi'$ update and queries
12             Add $(\Phi \wedge cdt, \Phi_F \wedge cdt, l_t)$ to $WL$
13         end
14     end
       /* Idem for else branch ($\neg cdt$)                    */
15 end
```

Both optimizations can be combined together as illustrated in Algorithm 9. Taking FASE-IOD as a basis, saturation detection is added in the faulted path predicate Φ_F queries at conditional branch handling. If the saturation is detected, the main path predicate switch to Φ_F but Φ_F stops being updated and queried further down that path, which stops fault injection.

Proposition 5. *The combination of FASE-EDS and FASE-IOD is correct and k-complete for the adversarial reachability problem.*

Proof. This combination also explores all possible paths for the considered attacker models, like FASE, hence preserving its properties.

6 Implementation

We now provide details about our forkless adversarial symbolic execution (FASE) implementation, named BINSEC/ASE, for Adversarial Symbolic Execution. *The code is made open-source*[7]*.*

[7] https://github.com/binsec/binsec-ase

Binary-level Fault Injection. While our method works for any program abstraction level, we choose to implement it for the binary level, which makes more sense in many security scenarios. We implement our forkless adversarial symbolic execution on top of the BINSEC symbolic engine [38,40,10]. It has already been used in a number of significant case studies [9,81,80,36,37], and it is notably able to achieve bounded verification (k-completeness) and to reasonably deal with symbolic pointers [44].

We modified the path predicate computation of BINSEC 0.4.0 as described in Section 5, and implemented our dedicated optimizations FASE-EDS, FASE-IOD and FASE EDS+IOD. BINSEC consists of 60kloc of Ocaml and our modifications add 6kloc. The attacker goal is specified as a local predicate to reach, using BINSEC directives. We currently support data faults such as arbitrary modification, bit-flip and reset. Test inversion is emulated through faulting the condition of conditional jumps. We let the user define an injection target range, made of multiple code address intervals. For large programs, it enables focusing on the security critical sections. Finally, we also provide a blacklist for some memory locations which will never be faulted. The blacklist is mostly used for the stack register (`esp` in x86, which is concretized in the analysis) and the program counter, as our fault model does not include tampering with the stack nor arbitrary control faults.

Details. Our exploration strategy is depth first, the underlying SMT solver is Bitwuzla [71]. We constrain the faulted values to differ from the original values in fault encodings, such that only true corruptions are reported as active faults.

7 Evaluation

We now evaluate our new algorithm for software verification against multi-fault attacks. We consider the following research questions.

- **RQ1**: is our tool correct and complete? In particular, can we find attacks on vulnerable programs and prove secure resistant programs?
- **RQ2**: can we scale in number of faults without path explosion?
- **RQ3**: what is the impact of our optimizations?

Besides this evaluation, we also show the use of our method in a number of different security scenarios (Section 7.5), and on a larger case study (Section 8).

7.1 Experimental Setting

The Machine Used. We ran our experiments on a cloud machine with a processor Intel Dual Xeon 4214R with 48 CPU cores and 384GB of RAM. Experiments ran in parallel on the 48 cores, each run using only one core.

The Attacker Model chosen in this evaluation can perform a varying number of faults. Its goal is expressed as a security oracle directly written in C for each benchmark, the computation of which is not faulted.

The Benchmark used here is a standard set of programs from the SWiFI literature on physical fault injections and high-security devices, characterized in Table 2. First, the 8 versions of VerifyPIN from the FISSC [42] benchmark suite, dedicated to the evaluation of physical fault attack analyses. VerifyPIN is an authentication program. There are one unprotected and 7 different protected versions, some vulnerable, some resistant to one test inversion fault. We added 2 manually unrolled versions of the unprotected VerifyPIN, with a PIN size of 4 and 16, to add diversity in the benchmarks with programs without loops. An oracle is provided by FISSC, checking if the user PIN truly corresponds to the reference PIN. Second, we take the 2 versions of the npo2 program from Le et al. [65], together with their oracles. Npo2 is a program computing an integer's upper power of two. The attacker's goal is to perform a silent data corruption, i.e. change the end result without triggering countermeasures. One version is vulnerable to one arbitrary data fault, the second is resistant due to extra arithmetic checks.

Compilation. The benchmarks are written in C and have been compiled with gcc for the Intel x86-32 architecture, using the flag "-O0" to preserve countermeasures. For BINSEC compatibility, we use the "-static" flag to include the necessary library functions directly in the binary.

Table 2: Benchmarks characteristics and statistics of a standard SE analysis

			BINSEC analysis - no fault			
Program group (#)	C loc	x86 loc	#instruction (explored)	#paths	#branch in a path	Time
Section 7						
VerifyPINs (8)	80-140	160-215	192-269	1	17-34	< 0.1s
VerifyPIN unrolled (2)	40-85	140-430	142-442	5-17	5-17	< 0.1s
npo2 (2)	50	200-220	607-653	3	31-33	< 0.1s
Section 8						
WooKey bootloader	3.2k	2350	290k	17	18k	9s
Section 7.5						
CRT-RSA (3)	125-170	400-600	108k-29M	1	5k-1.3M	0.4s - 1m27
Secret keeping machine (2)	100-200	240-360	1k-1.3k	1	130-150	< 0.1s
VerifyPIN_0 with SecSwift	80	430	430	1	22	< 0.1s

BINSEC Settings. We limit the maximal depth of an analysis to the depth necessary to perform an exhaustive non-faulty analysis, rounded to the upper hundred. We exhaustively explore all the possible paths up to this bound and do not stop at the first identified attack, in order to have comparable results. We set the global analysis timeout for 1 day. We fault values and not addresses, we do not directly fault the stack pointer nor the program counter, and we do not fault the status flags unless explicitly specified.

7.2 Correctness and Completeness in Practice (RQ1)

We first show that our tool works as expected on several codes with known ground truth. (1) We check that indeed, with no fault allowed, no attack is found in any of the benchmarks; (2) We check that indeed the insecure npo2 program is vulnerable to a single arbitrary data fault while the secure version is not – it can still be exploited with two faults; (3) According to their authors, the VerifyPIN versions 0 to 4 are vulnerable to one test inversion, while VerifyPIN 5 to 7 are resistant to it. We indeed reproduce these results. When allowing two faults, all VerifyPIN become vulnerable; (4) When using one arbitrary data fault against the VerifyPINs, all versions are found vulnerable. We manually check that indeed the identified attack paths make sense; (5) Our manually unrolled versions of VerifyPINs do not contain conditional branching instructions in the targeted function, making them resistant to test inversion. We check that this is the case, while they are still vulnerable to a single arbitrary data fault.

Conclusion. Our tool indeed can showcase a program vulnerability to fault injection attacks and prove resistance to fault injection attacks, as expected by the correctness and k-completeness properties of the underlying algorithms.

7.3 Scalability (RQ2)

For this evaluation, we focus on an attacker capable of arbitrary data faults, as those weigh the heaviest on the analysis.

We take FASE-IOD as our best performing technique (see Section 7.4). We evaluate here its capability to handle multi-fault and avoid path explosion, compared to the forking technique. Results are illustrated in Figures 4 and 5. Note that all FASE variants explore the same number of paths, and are thus represented as FASE in Figure 5. For each benchmark, we took the arithmetic mean for 100 runs. Values presented here are the geometric mean over the benchmarks.

FASE-IOD is 10x times faster than Forking for 1 fault, and x200 times faster for 2 faults on average. For the best case benchmark, we are x224 times faster for 1 fault and x6121 for 2. Starting from three faults onward, Forking experiences timeouts, rendering values non comparable. Half of the benchmark timeouts for 3 faults, three quarters for 4 faults, 11 over 12 for 6 faults and all of them after that. FASE-IOD never timeouts in this experiment. This scaling is enabled by avoiding path explosion. On average, Forking explores x50 times more paths for 2 faults than for one, while FASE-IOD only explores x3 times more paths. From Figure 4, we see FASE on its own already scales better than Forking, being x3 times faster for 1 fault and x108 times faster for 2, and never experiencing timeouts either.

Conclusion. FASE-IOD shows improved scalability in terms of the maximum number of faults allowed, for the arbitrary data fault model, compared to the forking technique.

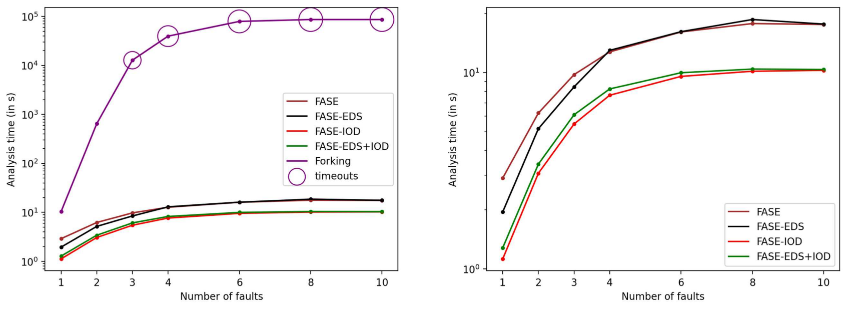


Fig. 4: Analysis time

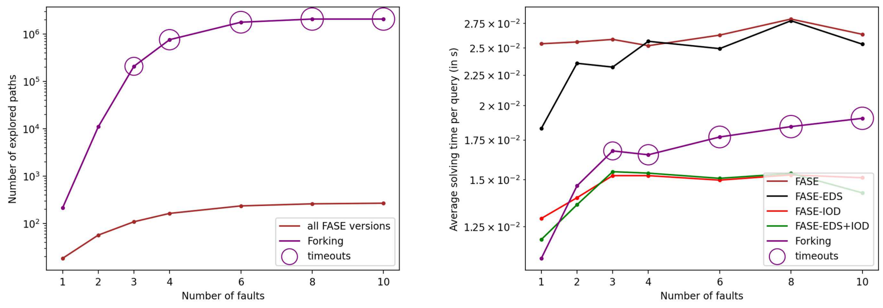


Fig. 5: Average number of explored paths, Average solving time per query

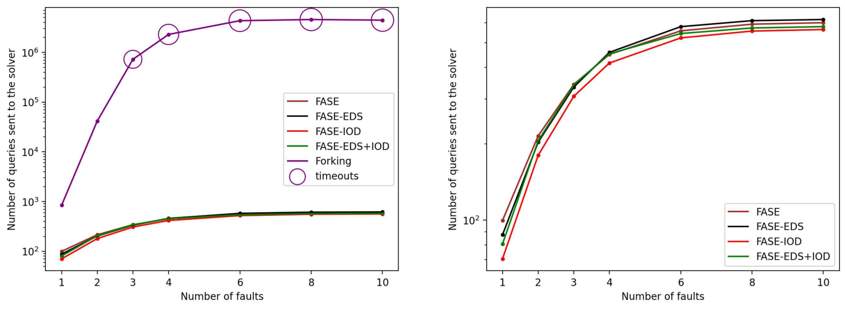


Fig. 6: Number of queries sent to the solver

7.4 Performance Optimization (RQ3)

We evaluate our forkless variants: FASE, FASE-EDS, FASE-IOD and FASE EDS+IOD, to determine which performs best for arbitrary data faults. Results are illustrated in Figures 4, 5 and 6.

We vary again the maximum number of faults from 1 to 10. Note that all FASE variants explore the same number of paths for each number of faults, as the optimizations reduce the number of faults injected but do not lose correctness nor k-completeness. FASE indeed generates complex queries[8], taking on average around twice the time necessary for Forking queries to be solved. FASE-EDS then gains a little bit in that regard. FASE queries take only x1.04 longer to solve on average for all fault numbers. The real improvement comes with the On-Demand logic of FASE-IOD (x2.02 times faster on average over all fault numbers) and FASE EDS+IOD (x2.02 also), where query complexity drops to the level of Forking. This improvement in query complexity is achieved algorithmically at the price of query creation. However, due to more queries being arithmetically simplified, fewer queries are sent in the end to the solver for FASE-IOD (x0.88 on average over all fault values compared with FASE) and FASE EDS+IOD (x0.98). FASE-EDS sent approximately the same number of queries as FASE. The number of queries sent to the solver explodes for Forking, correlated with the path explosion experienced. In terms of performance, two trends appear as the number of faults allowed increases. FASE and FASE-EDS tend to be between x2 and x3 times slower than FASE-IOD and FASE EDS+IOD. In the end, FASE-IOD proves to be the fastest optimization (x1.1 times faster than FASE EDS+IOD on average over all number of faults), likely due to the combination of on-demand logic and fewer queries than FASE EDS+IOD.

Conclusion. We retain FASE-IOD as our best performing forkless adversarial algorithm, at most x3.06 faster than FASE.

7.5 Other Experiments and Fault Models

CRT-RSA. Puys et al. [78] describe three versions of CRT-RSA: unprotected, Shamir version and Aumuller version. Only the last one is shown to resist the BellCoRe attack [16] which uses a single reset fault to break the cryptography. We were able to automatically reproduce the attack with 1 reset fault on the unprotected version of CRT-RSA, after 3s of analysis, and we were not able to find attacks on the other two versions in 10 days time.

Secret-keeping Machine. Dullien [41] proposes two versions of a secret-keeping machine. The one based on linked lists is manually shown to be exploitable by an attacker able to perform a single bit-flip in the memory (not in registers), while the array version is shown to be secure against that. For this benchmark,

[8] When counting the number of ite operators introduced in queries, from having barely any in a run without faults, we reach around 2,800 ite per query on average for FASE and 1,500 for FASE-IOD for one fault.

we activated faults on variables used as addresses. We were able to reproduce the attack on the linked list implementation with one bit-flip fault and to show the array implementation is secure for this fault model. In addition, if we allow faults in registers too, the array implementation becomes vulnerable.

SecSwift Countermeasure. We applied the SecSwift countermeasure, a llvm-level protection developed by STMicroelectronics [45,27], to VerifyPIN version 0. We were able to find attacks yielding an early loop exit on this binary with either a single test inversion or a single arbitrary data fault. These paths belonging to the CFG of the program, these attacks are not unexpected, yet it is still interesting that our method finds them automatically.

8 Case Study: the WooKey Bootloader

We now confront our tool to a real-life security system, WooKey.

Presentation of WooKey. First presented in 2018 by ANSSI, the French system security agency, the WooKey platform [14,89] is “a custom STM32-based USB thumb drive with mass storage capabilities designed for user data encryption and protection, with a full-fledged set of in-depth security defenses”. Their choice to be open source and open hardware makes WooKey a relevant case study: it is a real-life, complex device, security focused and available for reproducibility. Note also that Wookey has been extensively analyzed, as it was the target of an ANSSI cybersecurity challenge for security professionals [5].

Security Scenario and Goal of our Study. We focus on WooKey bootloader, a dual-bank system enabling hot firmware updates. The system is hardened, especially redundant test protections are present in critical sections to protect against test inversion faults. We consider the same attacker model as the ANSSI challenge did [5]: the attacker seeks to manipulate the bootloader logic to boot on the older firmware, more likely to contain security vulnerabilities. We also consider an attacker able to perform a single arbitrary data fault. We see in Table 2 that WooKey bootloader size is orders of magnitude larger than the programs used for evaluation in Section 7. Wookey is available as C code. We compile it like we did for the evaluation benchmarks (Section 7.1).

We conduct the following three analyses:

1. automatically analyze WooKey at binary-level to check whether we are able to find previously known faults [63], and/or new ones: we are indeed able to find the two faults identified by prior work [63] (A1, A2), *as well as an attack they do not mention* (A3);
2. automatically analyze at binary-level the patch version of Wookey proposed by Lacombe et al. [63]: we found that the proposed patch indeed blocks the two known attacks (A1 and A2), but not the new attack (A3);
3. propose a definitive patch by adding a counter-measure for A3 and remove parts of the counter-measures which are shown to be useless here. The patch is proven correct w.r.t. our attack model.

We discuss these results in the following and we present briefly in Section 8 the discovery of two more known faults. Overall, it demonstrates that our technique can scale to binary-level real-size systems.

Analyze Key Parts of Wookey. Lacombe et al. find an attack in the *loader_exec_req_selectbank* function (A1) and another in the *loader_exec_req_flashlock* function (A2). They correspond to data corruption in branching conditions. We are able to find both attacks, linking faults back to their locations in the C code with debug information. *We also find an additional attack*, faulting another part of the *loader_exec_req_flashlock* function (A3).

Analyze a Security Patch of WooKey. We now evaluate the protection scheme proposed by Lacombe et al. [63] for these attacks. It consists of four extra counter-measures named from CM1 to CM4. We found indeed that the full protection prevents attacks A1 and A2, as claimed by the authors of the patch. Yet, our analysis shows that the protection does not prevent the new attack A3.

Propose a New Patch and Evaluate It. We manually inspect these different analysis results to understand what happens. We have especially been able to identify the root cause of A3 and propose a dedicated countermeasure for it (named CMA). Also, by analyzing each counter-measure in isolation, we have been able to understand that counter-measures CM1 and CM3 do not block any attack path as they are redundant with other tests in the code and can be safely removed. Overall, our new patch (CMA + refined former patch) is shown by our tool to protect against all the attacks, for an attacker able to perform one arbitrary data fault (Table 3).

Table 3: Table summarizing the effects of countermeasures

Protection scheme	A1	A2	A3 (new)
	l.3	l.31	l.25
Normal Wookey	✓	✓	✓
Prior patch (CM1+CM2+CM3+CM4)	✗	✗	✓
Our patch (CM2+CM4+CMA)	✗	✗	✗

Legend - ✓: attack path found by our tool / ✗: no attack found

Other Attacks on WooKey. We were also able to find two other known attacks on Wookey. ***(Attack vector combination)*** The iso8716 library, used in WooKey for secure communication, presents a vulnerability to fault injection which enables a software buffer-overflow in function *SC_get_ATR* [63]. Using an attacker with a single arbitrary data fault, we were able to reproduce this attack; ***(Faulty redundant test)*** Martin et al. [68] shows an incorrect implementation of a redundant test to prevent single test inversion faults in the *loader_set_state* function. We reproduce this result.

9 Discussion

Fault Models. Our current approach does not support advanced *control* faults such as instruction corruption or instruction skip. Instruction corruption is out of scope as it permanently changes an instruction, while we modify computation results. It is related to self-modification, a notoriously difficult point to address in adversarial binary-level code analysis [17,77]. Instruction skip (or other arbitrary control jumps) could be modeled by local modification of the program counter, yet at the price of a huge path explosion. Also, regarding micro-architectural attacks, modeling Spectre attacks is difficult due to the speculative windows mechanism and its associated rollback.

Other Formal Methods. While in the paper we focus on symbolic execution, we believe the main optimization ideas developed here can be used with other formal techniques, e.g. Bounded Model Checking [29,31], Abstract Interpretation [34] or CEGAR [30]. Note that for each of them, fault injection may result either in path explosion or precision loss. Still, our forkless encoding should be able to help at least all approaches based to some extent on path unrolling.

Other Properties. The forkless encoding can surely benefit other classes of properties to be achieved by the attacker, especially those known to be supported by (extensions of) symbolic execution, for example: trace properties such as use-after-free, k-hyperreachability properties (secret leakage, privacy leakage, violation of constant-time, etc.) [36], the recent robust reachability proposal [48] for replicable bugs, etc. Our formalism itself is quite generic and can accommodate a wide range of properties, as we mainly keep the property unchanged but modify the underlying transition system. We could for example imagine an attacker willing to activate a non-terminating execution (denial of service).

Forkless Encoding and Instrumentation. Several prior works use code-level instrumentation [68] or LLVM-level instrumentation [76,63,65] in order to leverage standard program analyzers as is. The forkless encoding we propose can also be used this way, for more flexibility but without additional optimizations. Actually, we performed some experiments with Klee and a C-level forkless instrumentation, and do observe significant improvement over forking instrumentation.

10 Related Work

SWiFI. Prior work in SWiFI has already been discussed in Section 3. All methods in this domain consider low-level formalism: C [28,68], LLVM [76,63], binary [25,15,20,50]. Half of the techniques rely on the mutant approach [28,79,49,25,50], and the other half relies on forking [76,15,20,63]. While most approaches target attack finding (with symbolic execution and bounded model-checking), some do aim at full verification [79], especially with deductive verification [68,28]. Very few works consider multi-faults [76,63,68]. Interestingly, Lacombe et al. [63] propose a static way of reducing injection points on C programs, that is complementary to our own method – still, static analysis at binary-level is known to be hard. Note that a few methods do consider instruction skips [49,20,50], yet with path explosion issues.

Robustness Analysis. SWiFI is also used for robustness evaluation [64,74,56,88,65,72,32,90], in order to verify the correct behavior of error handling mechanisms. They rely also on forking or mutant techniques. The fault models are similar to hardware fault injection, yet multi-fault is not really an issue there, as faults are supposed to originate from safety issues (e.g. cosmic rays) and have no reason to accumulate unreasonably.

Formalizations and Fault Models. While it is common in the field of automated formal verification of cryptographic protocols to consider models of attackers (typically, extensions of the "Dolev-Yao" model) – either by specifying what the attackers can do [2] or what they cannot do [7], only very few formalizations of software-level attacker capabilities have been proposed so far. In software security, control-flow integrity attacks have been categorized by the capability an attacker needs [21], but these efforts have been restricted to manual reasoning. Interestingly, Given-Wilson et al. [51] propose a formalization of fault injection using Turing machines, but to our knowledge, no algorithm has been built for it. Also, Fournet et al. [46] propose a type system for program-level non-interference, taking into account an active adversary modeled as adversarial components able to perform any action at certain steps of the program.

Mutation Testing. Sometimes called software fault injection, mutation testing [75,33] aims to generate a comprehensive test suite by building test cases discriminating various mutants of a program, and is recognized as a very powerful testing criterion. As it focuses on coverage, mutant explosion cannot be avoided. Dedicated SE techniques [73,8,11,67] have been designed.

11 Conclusion

We formalize the concept of adversarial reachability, extending standard reachability to include the presence of an advanced attacker in program analysis, and we propose a dedicated symbolic algorithm for adversarial reachability, integrating a novel forkless encoding of faults together with dedicated optimizations. Our technique is shown to significantly reduce the number of paths to explore, and scales up to 10 faults on a standard SWiFI benchmark, where prior forking attempts timeout for 3 faults. Also, we show that our method scale to realistic size examples, such as the WooKey project where we have been able to replay known fault attacks and to even find a vulnerability not mentioned in a recently proposed countermeasure patch.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Transactions on Information and System Security (TISSEC) **13**(1), 1–40 (2009)
2. Abadi, M., Cortier, V.: Deciding knowledge in security protocols under equational theories. In: International Colloquium on Automata, Languages, and Programming. pp. 46–58. Springer (2004)
3. Akhunzada, A., Sookhak, M., Anuar, N.B., Gani, A., Ahmed, E., Shiraz, M., Furnell, S., Hayat, A., Khan, M.K.: Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. Journal of Network and Computer Applications **48**, 44–57 (2015)
4. Anceau, S., Bleuet, P., Clédière, J., Maingault, L., Rainard, J.l., Tucoulou, R.: Nanofocused X-ray beam to reprogram secure circuits. In: International Conference on Cryptographic Hardware and Embedded Systems. pp. 175–188. Springer (2017)
5. ANSSI, Amossys, EDSI, LETI, Lexfo, Oppida, Quarkslab, SERMA, Synacktiv, Thales, Labs, T.: Inter-cesti: Methodological and technical feedbacks on hardware devices evaluations. In: SSTIC 2020, Symposium sur la sécurité des technologies de l'information et des communications (2020)
6. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In: International Conference on Integrated Formal Methods. pp. 1–20. Springer (2004)
7. Bana, G., Comon-Lundh, H.: A computationally complete symbolic attacker for equivalence properties. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 609–620 (2014)
8. Bardin, S., Chebaro, O., Delahaye, M., Kosmatov, N.: An all-in-one toolkit for automated white-box testing. In: International Conference on Tests and Proofs. pp. 53–60. Springer (2014)
9. Bardin, S., David, R., Marion, J.Y.: Backward-bounded dse: targeting infeasibility questions on obfuscated codes. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 633–651. IEEE (2017)
10. Bardin, S., Herrmann, P., Leroux, J., Ly, O., Tabary, R., Vincent, A.: The bincoa framework for binary code analysis. In: International Conference on Computer Aided Verification. pp. 165–170. Springer (2011)
11. Bardin, S., Kosmatov, N., Cheynier, F.: Efficient leveraging of symbolic execution to advanced coverage criteria. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. pp. 173–182. IEEE (2014)
12. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of model checking, pp. 305–343. Springer (2018)
13. Barthe, G., Dupressoir, F., Fouque, P.A., Grégoire, B., Zapalowicz, J.C.: Synthesis of fault attacks on cryptographic implementations. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1016–1027 (2014)
14. Benadjila, R., Renard, M., Trebuchet, P., Thierry, P., Michelizza, A., Lefaure, J.: Wookey: Usb devices strike back. Proceedings of SSTIC (2018)
15. Berthier, M., Bringer, J., Chabanne, H., Le, T.H., Rivière, L., Servant, V.: Idea: embedded fault injection simulator on smartcard. In: International Symposium on Engineering Secure Software and Systems. pp. 222–229. Springer (2014)
16. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: International conference on the theory and applications of cryptographic techniques. pp. 37–51. Springer (1997)

17. Bonfante, G., Fernandez, J., Marion, J.Y., Rouxel, B., Sabatier, F., Thierry, A.: Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 745–756 (2015)
18. Bounimova, E., Godefroid, P., Molnar, D.: Billions and billions of constraints: Whitebox fuzz testing in production. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 122–131. IEEE (2013)
19. Bozzato, C., Focardi, R., Palmarini, F.: Shaping the glitch: optimizing voltage fault injection attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 199–224 (2019)
20. Bréjon, J.B., Heydemann, K., Encrenaz, E., Meunier, Q., Vu, S.T.: Fault attack vulnerability assessment of binary code. In: Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems. pp. 13–18 (2019)
21. Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M.: Control-flow integrity: Precision, security, and performance. ACM Computing Surveys (CSUR) **50**(1), 1–33 (2017)
22. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
23. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: Automatically generating inputs of death. ACM Transactions on Information and System Security (TISSEC) **12**(2), 1–38 (2008)
24. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Communications of the ACM **56**(2), 82–90 (2013)
25. Carré, S., Desjardins, M., Facon, A., Guilley, S.: Openssl bellcore's protection helps fault attack. In: 2018 21st Euromicro Conference on Digital System Design (DSD). pp. 500–507. IEEE (2018)
26. Cervesato, I.: The dolev-yao intruder is the most powerful attacker. In: 16th Annual Symposium on Logic in Computer Science—LICS. vol. 1, pp. 1–2. Citeseer (2001)
27. Chauvet, H., de Ferrière, F., Bizet, T.: Software fault injection for secswift qualification (2021), https://jaif.io/2021/media/JAIF2021%20-%20deFerriere.pdf
28. Christofi, M., Chetali, B., Goubin, L.: Formal verification of an implementation of crt-rsa vigilant's algorithm. In: PROOFS workshop: pre-proceedings. vol. 28 (2013)
29. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. (2001)
30. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM (2003)
31. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ansi-c programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 168–176. Springer (2004)
32. Cotroneo, D., De Simone, L., Liguori, P., Natella, R.: Profipy: Programmable software fault injection as-a-service. In: 2020 50th annual IEEE/IFIP international conference on dependable systems and networks (DSN). pp. 364–372. IEEE (2020)
33. Cotroneo, D., Natella, R.: Fault injection for software certification. IEEE Security & Privacy **11**(4), 38–45 (2013)
34. Cousot, P.: Abstract interpretation. ACM Computing Surveys (CSUR) **28**(2), 324–328 (1996)
35. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: Programming Languages and Systems (2005)

36. Daniel, L.A., Bardin, S., Rezk, T.: Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1021–1038. IEEE (2020)
37. Daniel, L.A., Bardin, S., Rezk, T.: Hunting the haunter-efficient relational symbolic execution for spectre with haunted relse. In: NDSS (2021)
38. David, R., Bardin, S., Ta, T.D., Mounier, L., Feist, J., Potet, M.L., Marion, J.Y.: Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In: SANER (2016)
39. Dehbaoui, A., Dutertre, J.M., Robisson, B., Tria, A.: Electromagnetic transient faults injection on a hardware and a software implementations of AES. In: 2012 Workshop on Fault Diagnosis and Tolerance in Cryptography. pp. 7–15. IEEE (2012)
40. Djoudi, A., Bardin, S.: Binsec: Binary code analysis with low-level regions. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 212–217. Springer (2015)
41. Dullien, T.: Weird machines, exploitability, and provable unexploitability. IEEE Transactions on Emerging Topics in Computing **8**(2), 391–403 (2017)
42. Dureuil, L., Petiot, G., Potet, M.L., Le, T.H., Crohen, A., Choudens, P.d.: Fissc: A fault injection and simulation secure collection. In: International Conference on Computer Safety, Reliability, and Security. pp. 3–11. Springer (2016)
43. Facebook: Infer static analyzer. https://fbinfer.com/
44. Farinier, B., David, R., Bardin, S., Lemerre, M.: Arrays made simpler: An efficient, scalable and thorough preprocessing. In: LPAR. pp. 363–380 (2018)
45. de Ferrière, F.: Software countermeausres in the llvm risc-v compiler (2021), https://open-src-soc.org/2021-03/media/slides/3rd-RISC-V-Meeting-2021-03-30-15h00-Fran%C3%A7ois-de-Ferri%C3%A8re.pdf
46. Fournet, C., Rezk, T.: Cryptographically sound implementations for typed information-flow security. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. ACM (2008)
47. Gangolli, A., Mahmoud, Q.H., Azim, A.: A systematic review of fault injection attacks on iot systems. Electronics **11**(13), 2023 (2022)
48. Girol, G., Farinier, B., Bardin, S.: Not all bugs are created equal, but robust reachability can tell the difference. In: International Conference on Computer Aided Verification. pp. 669–693. Springer (2021)
49. Given-Wilson, T., Jafri, N., Lanet, J.L., Legay, A.: An automated formal process for detecting fault injection vulnerabilities in binaries and case study on present. In: 2017 IEEE Trustcom/BigDataSE/ICESS. pp. 293–300. IEEE (2017)
50. Given-Wilson, T., Jafri, N., Legay, A.: Combined software and hardware fault injection vulnerability detection. Innovations in Systems and Software Engineering **16**(2), 101–120 (2020)
51. Given-Wilson, T., Legay, A.: Formalising fault injection and countermeasures. In: Proceedings of the 15th International Conference on Availability, Reliability and Security. pp. 1–11 (2020)
52. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. pp. 213–223 (2005)
53. Godefroid, P., Levin, M.Y., Molnar, D.: Sage: whitebox fuzzing for security testing. Communications of the ACM **55**(3), 40–44 (2012)

54. Goyal, B., Sitaraman, S., Venkatesan, S.: A unified approach to detect binding based race condition attacks. In: Int'l Workshop on Cryptology & Network Security (CANS). p. 16 (2003)
55. Gravellier, J., Dutertre, J.M., Teglia, Y., Moundi, P.L.: Faultline: Software-based fault injection on memory transfers. In: 2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 46–55. IEEE (2021)
56. Hari, S.K.S., Tsai, T., Stephenson, M., Keckler, S.W., Emer, J.: Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In: 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 249–258. IEEE (2017)
57. Van den Herrewegen, J., Oswald, D., Garcia, F.D., Temeiza, Q.: Fill your boots: Enhanced embedded bootloader exploits via fault injection and binary analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 56–81 (2021)
58. Karaklajić, D., Schmidt, J.M., Verbauwhede, I.: Hardware designer's guide to fault attacks. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **21**(12), 2295–2306 (2013)
59. Kim, C.H., Quisquater, J.J.: Fault attacks for CRT based RSA: New attacks, new results, and new countermeasures. In: IFIP International Workshop on Information Security Theory and Practices. pp. 215–228. Springer (2007)
60. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. Form. Asp. Comput. (2015)
61. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al.: Spectre attacks: Exploiting speculative execution. In: SP (2019)
62. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al.: Spectre attacks: Exploiting speculative execution. Communications of the ACM **63**(7), 93–101 (2020)
63. Lacombe, G., Feliot, D., Boespflug, E., Potet, M.L.: Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities. In: PROOFS WORKSHOP (SECURITY PROOFS FOR EMBEDDED SYSTEMS) (2021)
64. Larsson, D., Hähnle, R.: Symbolic fault injection. In: International Verification Workshop (VERIFY). vol. 259, pp. 85–103. Citeseer (2007)
65. Le, H.M., Herdt, V., Große, D., Drechsler, R.: Resilience evaluation via symbolic fault injection on intermediate code. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 845–850. IEEE (2018)
66. Le, Q.L., Raad, A., Villard, J., Berdine, J., Dreyer, D., O'Hearn, P.W.: Finding real bugs in big programs with incorrectness logic. Proceedings of the ACM on Programming Languages **6**(OOPSLA1), 1–27 (2022)
67. Marcozzi, M., Bardin, S., Kosmatov, N., Papadakis, M., Prevosto, V., Correnson, L.: Time to clean your test objectives. In: Proceedings of the 40th International Conference on Software Engineering. pp. 456–467 (2018)
68. Martin, T., Kosmatov, N., Prevosto, V.: Verifying redundant-check based countermeasures: a case study. In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. pp. 1849–1852 (2022)
69. Murdock, K., Oswald, D., Garcia, F.D., Van Bulck, J., Gruss, D., Piessens, F.: Plundervolt: Software-based fault injection attacks against intel sgx. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1466–1482. IEEE (2020)

70. Mutlu, O., Kim, J.S.: Rowhammer: A retrospective. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(8), 1555–1571 (2019)
71. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR **abs/2006.01621** (2020), https://arxiv.org/abs/2006.01621
72. Palazzi, L., Li, G., Fang, B., Pattabiraman, K.: A tale of two injectors: End-to-end comparison of ir-level and assembly-level fault injection. In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). pp. 151–162. IEEE (2019)
73. Papadakis, M., Malevris, N.: Automatic mutation test case generation via dynamic symbolic execution. In: 2010 IEEE 21st International Symposium on Software Reliability Engineering. pp. 121–130. IEEE (2010)
74. Pattabiraman, K., Nakka, N., Kalbarczyk, Z., Iyer, R.: Symplfied: Symbolic program-level fault injection and error detection framework. In: 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN). pp. 472–481. IEEE (2008)
75. Petrovic, G., Ivankovic, M., Kurtz, B., Ammann, P., Just, R.: An industrial application of mutation testing: Lessons, challenges, and research directions. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 47–53. IEEE (2018)
76. Potet, M.L., Mounier, L., Puys, M., Dureuil, L.: Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. pp. 213–222. IEEE (2014)
77. Preda, M.D., Giacobazzi, R., Debray, S., Coogan, K., Townsend, G.M.: Modelling metamorphism by abstract interpretation. In: International Static Analysis Symposium. pp. 218–235. Springer (2010)
78. Puys, M., Riviere, L., Bringer, J., Le, T.h.: High-level simulation for multiple fault injection evaluation. In: Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance, pp. 293–308. Springer (2014)
79. Rauzy, P., Guilley, S.: A formal proof of countermeasures against fault injection attacks on crt-rsa. Journal of Cryptographic Engineering **4**(3), 173–185 (2014)
80. Recoules, F., Bardin, S., Bonichon, R., Lemerre, M., Mounier, L., Potet, M.L.: Interface compliance of inline assembly: Automatically check, patch and refine. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 1236–1247. IEEE (2021)
81. Recoules, F., Bardin, S., Bonichon, R., Mounier, L., Potet, M.L.: Get rid of inline assembly through verification-oriented lifting. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 577–589. IEEE (2019)
82. Richter-Brockmann, J., Sasdrich, P., Guneysu, T.: Revisiting fault adversary models–hardware faults in theory and practice. IEEE Transactions on Computers (2022)
83. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for c. ACM SIGSOFT Software Engineering Notes **30**(5), 263–272 (2005)
84. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security and Privacy (2016)

85. Skorobogatov, S.P., Anderson, R.J.: Optical fault induction attacks. In: International workshop on cryptographic hardware and embedded systems. pp. 2–12. Springer (2002)
86. Tang, A., Sethumadhavan, S., Stolfo, S.: {CLKSCREW}: Exposing the perils of {Security-Oblivious} energy management. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 1057–1074 (2017)
87. Van Bulck, J., Moghimi, D., Schwarz, M., Lippi, M., Minkin, M., Genkin, D., Yarom, Y., Sunar, B., Gruss, D., Piessens, F.: Lvi: Hijacking transient execution through microarchitectural load value injection. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 54–72. IEEE (2020)
88. Winter, S., Tretter, M., Sattler, B., Suri, N.: simfi: From single to simultaneous software fault injections. In: 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 1–12. IEEE (2013)
89. https://github.com/wookey-project, accessed july 2021
90. Zavalyshyn, I., Given-Wilson, T., Legay, A., Sadre, R., Riviere, E.: Chaos duck: A tool for automatic iot software fault-tolerance analysis. In: 2021 40th International Symposium on Reliable Distributed Systems (SRDS). pp. 46–55. IEEE (2021)

Automated Grading of Regular Expressions

Su-Hyeon Kim[1,3], Youngwook Kim[2], Yo-Sub Han[2], Hyeonseung Im[1], and Sang-Ki Ko[1](✉)

[1] Department of Computer Science & Engineering, Kangwon National University, Gangwon-do 24341, Republic of Korea
{tngus98207,hsim,sangkiko}@kangwon.ac.kr

[2] Department of Computer Science, Yonsei University, Seoul 03722, Republic of Korea
{youngwook,emmous}@yonsei.ac.kr

[3] Artificial Intelligence Research Center, Korea Electronics Technology Institute, Seongnam-si 13509, Republic of Korea
suhyeon0123@keti.re.kr

Abstract. With the rapid transition to distance learning, automatic grading software becomes more important to both teachers and students. We study the problem of automatically grading the regular expressions submitted by students in courses related to automata and formal language theory. In order to utilize the semantic information of the regular expression, we define a declarative logic that can be described by regular language and at the same time has natural language characteristics, and use it for the following tasks: 1) to assign partial grades for incorrect regular expressions and 2) to provide helpful feedback to students to make them understand the reason for the grades and a way to revise the incorrect regular expressions into correct ones. We categorize the cases when students' incorrect submissions deserve partial grades and suggest how to assign appropriate grades for each of the cases. In order to optimize the runtime complexity of the algorithm, two heuristics based on automata theory are proposed and evaluated on the dataset collected from undergraduate students. In addition, we suggest Regex2NL which translates regular expressions to natural language descriptions to give insight to students so that they can understand how the regular expressions work.

Keywords: regular expressions · MSO logic · automated grading system · automata theory

1 Introduction

Regular expressions (regexes) are a great tool for the pattern matching problem as they can effectively describe pattern structures. Regexes are widely used in software applications such as search engines, text processing, programming languages, and compilers due to their compact representations. Although most developers find that regexes are powerful and flexible tools, they also feel that regexes are very difficult to learn for many reasons such as readability, validity, reliability, and so on [7,16].

There have been several interesting approaches to automatically grading student submissions in an automata-related course in the online education environment. Alur et al. [2] propose a technique for automatically grading students' DFA construction in automata courses while generating high-level hints for helping students understand how to correct their wrong submissions. For instance, they introduce the DFA edit difference to compute the amount of difference between the correct DFA and students' DFA and MOSEL (MSO-equivalent declarative logic) even to capture the case where the student's submission corresponds to a different logic in MOSEL. Later, D'Antoni et al. [6] utilize the DFA edit difference in order to generate natural language feedback explaining how to correct the submitted DFA. They also conduct an online survey to collect students' feedback about the quality, usability, and effectiveness of their grading system.

Kakkar [10] studies a similar problem, namely, the problem of grading regexes instead of DFAs. Inspired by the DFA edit difference [2], Kakkar proposes a new criterion called 'Regex Edit Distance' which is basically based on the string edit-distance between students' regexes and correct ones. However, both works suffer from a limitation that 'optimal' answers for the problems should be given by TAs as they compare the students' submissions with the answers for giving partial grades. Recently, D'Antoni et al. [5] propose Automata Tutor v3 (abbreviated to AT v3 hereafter), which is the latest version of the previous work [2]. In AT v3, they include automated grading and feedback generation for a variety of new automata problems including the problems that ask to create regexes, context-free grammars, pushdown automata, and even Turing machines for a given description (e.g., a natural language description, or an automaton, or a grammar that belongs to a different class). However, they also rely on the string edit-distance for grading regexes similar to the work of [10]. Note that AT v3 provides counterexamples of incorrect regexes such as strings that should (or not) be accepted by students as feedback.

In this paper, we introduce an automated grading framework for regular expressions that gives reasonable grades and helpful feedback. The overall structure of our regex grading scheme is illustrated in Fig. 1. As the regex construction problem's goal is to make regex from the natural language description, TA first assigns the problem by giving the natural description of the problem and the logic formula of the regex which is one of the forms of the regular language. Then students submit the regex corresponding to the given description. Finally, we use three algorithms for generating more convincing partial grades and feedback by comparing the answer logic formula with the submission.

We aim to overcome several remaining limitations that have not been resolved by the earlier approaches. First, we claim that it is not appropriate to grade a student's regex just by calculating the string edit-distance with the 'solution regex'. There could be infinitely many regexes that describe the same language. Even when we consider the set of most compact regexes describing the regular

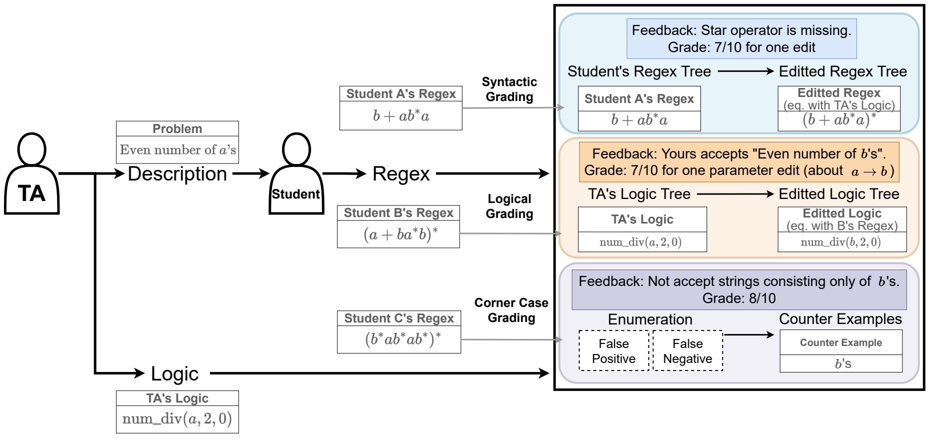


Fig. 1. Overview of our automated regex grading framework

language in question, there can be multiple regexes since it is not guaranteed that there is a unique minimal regex for a given regular language. Also, the string edit-distance cannot take the structural similarity into account while we can obtain hierarchical information from the tree form of the regex. Second, we should consider not only the syntactic discrepancies but also the semantic discrepancies arising from the misinterpretation of the problem. In order to compare the logical differences in real-time, the regex must be transformed with the logic and converted to DFA in polynomial time. However, there is no compact logic to do so. Lastly, there is a lack of abundant feedback that helps students study regexes. More detailed feedback such as suggesting the shortest form of the regex, logical differences between the answer and the submission, and organized form of the corner case would be more helpful than simple symbol correction feedback.

In order to resolve the above-mentioned issues, we propose a 3-step regex grading scheme that considers both syntactic and semantic discrepancies between submitted regexes and answer logic formulas (natural language descriptions). More specifically, first, to consider the syntactic discrepancy, instead of comparing a student's regex with the solution regex, we compare the possible transforms of the student's regex with the language of the solution. To this end, we apply tree-level edits to the parse tree of the regex to detect the possible syntactic mistakes made by the student. As shown in Fig. 1, after one tree-edit with adding the star operator to student A's submission $b+ab^*a$, the edited regex is equivalent to TA's logic $(b + ab^*a)^*$. Second, we take into account the possibility that a student simply misinterprets the specification of the language. For instance, we may consider that a submitted regex deserves a partial grade if the language expressed by the submission corresponds to a specification that is very similar to the given specification. Therefore, we consider the semantic discrepancy by applying logic-level edits to the logic formula for the specification and searching for a similar specification that exactly corresponds to the student's regex. In this way, by considering the 'similarity' to the student's regex, we can give a partial grade. For example, after one logic-edit with changing the parameter from 'a' to 'b' on the TA's logic, edited logic $\text{num\_div}(b, 2, 0)$ is equivalent to the student B's submission $(a + ba^*b)^*$. Finally, we take some corner cases into accounts such as when the language of a submitted regex misses a reasonably small portion of the target language such as the empty string or a language consisting of a single symbol (a^* or b^* when $\Sigma = \{a, b\}$). For instance, we can find that $(b^*ab^*ab^*)^*$ cannot generate strings that have zero number of a's and at least one b while it generates the empty string. Moreover, we generate productive feedback for students using the byproduct of each partial grading algorithm so that they can understand what is wrong with the current submission and how to correct the submission into a correct regex.

The rest of the paper is organized as follows. Section 2 gives some definitions and notations. We introduce a set of declarative logic formulas for describing regular languages in Section 3 and our regex grading scheme in Section 4. The experimental results are provided in Section 5 and Section 6 concludes the paper.

2 Preliminaries

The size of a finite set S is denoted by $|S|$. Let Σ denote a finite alphabet and Σ^* denote the set of all finite strings over Σ. For $m \in \mathbb{N}$, $\Sigma^{\leq m}$ is the set of strings of length at most m over Σ. A language over Σ is a subset of Σ^*. Given a set X, 2^X denotes the power set of X. The symbol λ denotes the empty string. We define $\text{mod}(m, n)$ to be $\{k \mid k \mod m = n, k \in \mathbb{N}\}$. We also define $\text{ind}(w, x) = \{k \mid w[k : k + |x|] = x, k \in \mathbb{N}\}$, where $w[i : j]$ for $i \leq j$ denotes a substring of w concatenating characters of w from index i to $j - 1$, to be the set of indices where x appears in w. Note that the index starts from 1.

A *regular expression* (regex) over Σ is $a \in \Sigma$ or the empty string λ, or is obtained by applying the following rules finitely many times. For regexes R_1 and R_2, the union $R_1 + R_2$, the concatenation $R_1 \cdot R_2$, and the Kleene-star R_1^* are also regexes.

Now we introduce a formal logic to be used to formally describe languages. Let $w = w_1 w_2 \cdots w_n$ be a word over Σ. For any $i \in [1, n]$ and a symbol $a \in \Sigma$, we say that a *letter predicate* a is true at i in w if $w_i = a$. For example, the logic formula $a(x) \land \exists y(y > x \land b(y))$ means that 'there is a symbol a at the position x and a symbol b at the position later than x'. It is readily seen that the formula describes the language described by the following regex: $a(a + b)^*b(a + b)^*$. It is well-known that regular languages are expressible in monadic second-order (MSO) logic [4].

Given a regex R, we define the parse tree $T(R)$ to be the rooted tree representing the hierarchical structure of R. Each leaf is labeled by a symbol in $\Sigma \cup \{\lambda\}$ and each internal node is labeled by n-ary operations such as $\cdot$ (concatenation) and $+$ (union), or unary operation $*$ (Kleene-star). We define the *regex tree edit-distance* $\text{ed}_{\text{rt}}(R, R')$ of two regexes R and R' to be the tree edit-distance between two parse trees of R and R'. Note that the tree edit-distance between

$T(R)$ and $T(R')$ is defined as the minimum number of edit-operations required to transform the tree $T(R)$ into $T(R')$, where an edit-operations for the regex tree edit-distance can be defined as a substitution of an operation symbol or a character from Σ into a different operation symbol (or a character from Σ), an insertion of a node, or a deletion of a node. It should be mentioned that we perform *unordered matching* between children of nodes labeled by the union $+$ operator as the order of elements inside the union operator does not matter.

3 Simple Declarative Logic for Regular Languages

Since MSO logic formulas offer a relatively higher-level specification of regular languages than finite-state automata recognizing the languages, they can be used for describing regular languages in a human-readable format. Moreover, we can always compile an MSO logic formula for a regular language into a corresponding minimal DFA [12] and therefore, a regex as well.

As the transformation from MSO to DFA may require the size of the alphabet to grow exponentially in the number of nested quantifiers [8], we restrict our attention to the logic formulas that can describe all regular languages considered in famous automata textbooks without covering the whole regular languages while being able to be converted into a corresponding DFA in polynomial time. Table 2 shows the list of declarative logic formulas considered in this paper. Recall that MOSEL [2], an extension of MSO logic with some syntactic sugar to allow describing regular languages more concisely, is introduced for a similar reason. However, we claim that our logic formulas directly correspond to NL descriptions at a much higher-level and allow us to perform language equivalence tests in practical runtime.

Analogously to the parse tree of a regex, we define the parse tree $T(\phi)$ for a given logic formula ϕ. Here each leaf is labeled by an atomic formula and each internal node is labeled by unary logical connectives $\neg$ (negation) or n-ary logical connectives such as $\wedge$ (conjunction) and $\vee$ (disjunction). Similarly to the regex tree edit-distance, we also define the *logic tree edit-distance* $\mathrm{ed}_{\mathrm{lt}}(\phi, \tilde{\phi})$ of two logic formulas ϕ and $\tilde{\phi}$ as the unordered tree edit-distance between two parse trees of ϕ and $\tilde{\phi}$. Note that we allow the substitution of an atomic logic formula and two logical connectives, conjunction, and disjunction, for the logic tree edit-distance. We also allow the insertion and deletion of negation. The substitution of an atomic logic formula is available for a single parameter such as strings x, y, non-negative integers m, n, and a comparison operator $\square \in \{>, =, <\}$. While the edit cost of the substitution of a logical connective equals 1, we assign the string edit-distance for the substitution of a string parameter, the numerical difference for an integer, and the value 1 for the substitution of a comparison operator.

We provide a list of regex problems and solutions collected from famous automata textbooks in Table 1. For each problem, we provide a natural language description for a regular language in question, a solution regular expression given in the textbook, and the corresponding logic formula found by us. We denote $a + \lambda$ by $a^?$ for brevity.

Table 1. A list of regex problems from famous automata textbooks.

No.	Description	Solution Regex	Logic Formula
1	Starts with a.	$a\sigma^*$	pos$(a, 1)$
2	Ends with ab	$.\ \sigma^* ab$	pos_rev$(ba, 1)$
3	Contains the substring $abab$.	$\sigma^* abab\sigma^*$	num$(abab, >, 0)$
4	Begins with b and ends with a.	$b\sigma^* a$	pos$(b, 1) \wedge$ pos_rev$(a, 1)$
5	Length is at least 3 and the 3rd symbol is a.	$\sigma\sigma a\sigma^*$	pos$(a, 3)$
6	Length is a multiple of 3.	$(\sigma\sigma\sigma)^*$	len_div$(a, 3, 0)$
7	The number of a's is divisible by 3	$(b^*ab^*ab^*ab^*)^*$.	num_div$(a, 3, 0)$
8	Even number of a's.	$(b + ab^*a)^*$	num_div$(a, 2, 0)$
9	The 5th symbol from the right end is b.	$\sigma^* b\sigma\sigma\sigma\sigma$	pos_rev$(b, 5)$
10	a and b alternate.	$b^?(ab)^*a^?$	num$(aa, =, 0) \wedge$num$(bb, =, 0)$
11	Each a is followed by at least one b.	$(a^?b)^*$	allX_followedbyY(a, b)
12	$a^n b^m$ where $n \geq 3$ and m is even	$aaaa^*(bb)^*$.	allX_beforeY(a, b) $\wedge$ num$(a, >, 2) \wedge$ num_div$(b, 2, 0)$
13	Contains less than three a's.	$b^*a^?b^*a^?b^*$	num$(a, <, 3)$
14	Start with a and have odd length or start with b and have even length.	$a(\sigma\sigma)^* + b\sigma(\sigma\sigma)^*$	(pos$(a, 1) \wedge$ len_div$(2, 1)) \vee$ (pos$(b, 1) \wedge$ len_div$(2, 0))$
15	Any strings except a and b.	$((\sigma\sigma)\sigma^*)^?$	$\neg$single_word(a) $\wedge$ $\neg$single_word(b)
16	Does not end with ab	$\sigma^*(aa + ba + bb) + \sigma^?$.	$\neg$pos_rev$(ba, 1)$
17	Contains at least one a and at most one b.	$aa^* + aa^*ba + a^*baa^*$	num$(a, >, 0) \wedge$ num$(b, <, 2)$
18	At least two occurrences of b between any two occurrences of a.	$b^* + b^*(abbb^*)^*ab^*$	exists_between$(b, a, 2)$
19	Does not contain baa as a substring.	$a^*(ba + b)^*$	num$(baa, =, 0)$
20	Every odd position is b.	$(b(\sigma b)^*\sigma^?)^?$	pos_every$(b, 2, 1)$
21	Has exactly one pair of consecutive a's.	$(ab + b)^*aa(ba + b)^*$	num$(aa, =, 1)$
22	Does not end with ba and the length is at least two.	$\sigma^*(aa + ab + bb)^*$	$\neg$pos_rev$(ab, 1) \wedge$ len$(1, >)$
23	Even number of a's and each a is followed by at least one b.	$b^*(abb^*abb^*)^*$ 24	num_div$(a, 2, 0)$ $\wedge$ allX_followedbyY(a, b)
25	Every pair of adjacent a's appears before any pair of adjacent b's.	$(a + ba)^*(b + ab)^*a^?$	allX_beforeY(aa, bb)
26	At most one pair of consecutive b's.	$(a + ba)^*(bb)^?(a + ab)^*$	num$(bb, <, 2)$

4 Regex Grading Algorithm

In this section, we explain our automated regex grading algorithm by considering both syntactic and semantic properties.

4.1 Grading of Regexes

Let us assume that exact logic formulas for regular languages asked in questions are already known as teachers always can specify the regular languages with

Table 2. A list of declarative logic formulas used to describe regular languages that appear in famous automata textbooks, where $m, n \in \mathbb{N}$, $a, b \in \Sigma$, $x, y \in \Sigma^*$, and $\Box \in \{>, =, <\}$. In the set notation, we broadcast $+n$ and $-n$ for some integer n to each element of the given set.

Logic Formula	**Description / Set Notation**
single_word(x)	Accepts a string x. / $\{x\}$
pos(x, n)	Substring x starts at nth position. / $\{wxv \mid \lvert w\rvert = n-1 \wedge w, v \in \Sigma^*\}$
pos_rev(x, n)	Substring x starts at nth position in reverse order. / $\{wxv \mid \lvert v\rvert = n-1 \wedge w, v \in \Sigma^*\}$
len($\Box, n$)	Strings of length $\Box\, n$. / $\{x \mid \lvert x\rvert \,\Box\, n\}$
len_div(m, n)	Strings of length $\in$ mod(m, n). / $\{x \mid \lvert x\rvert \in \text{mod}(m, n)\}$
pos_every(x, m, n)	Substring x appears at every mod(m, n)th position. / $\{w \mid \text{ind}(w, x) = \text{mod}(m, n) \cap [1, \lvert w\rvert]\}$
num($x, \Box, n$)	Contains x as a substring $\Box\, n$ times. / $\{w \mid \lvert\text{ind}(w, x)\rvert \,\Box\, n\}$
num_div(x, m, n)	Contains x as a substring mod(m, n) times. / $\{w \mid \lvert\text{ind}(w, x)\rvert \in \text{mod}(m, n)\}$
allX_followedbyY(x, y)	Every substring x is followed by y. / $\{w \mid \text{ind}(w, x) + \lvert x\rvert \subseteq \text{ind}(w, y)\}$
allX_followingY(x, y)	Every substring x is following y. / $\{w \mid \text{ind}(w, x) - \lvert y\rvert \subseteq \text{ind}(w, y)\}$
allX_beforeY(x, y, n)	Every substring x appears before any occurrence of y. / $\{w \mid \max(\text{ind}(w, x)) < \min(\text{ind}(w, y))\}$
exists_between(b, a, n)	b appears n times between every adjacent pair of a's. / $\{w \mid \forall i, j \in \text{ind}(w, a) \text{ s.t. } \lvert\text{ind}(w, a) \cap [i, j]\rvert = 2,$ $\lvert\text{ind}(w, b) \cap [i, j]\rvert = n\}$
consecutive($a, \Box, n$)	Every a appears $\Box\, n$ times consecutively. / $\{w \mid \forall i \in \text{ind}(w, a) \text{ s.t. } w[i-1] \neq a \text{ and}$ $j = \arg\max_j \{w[i:j] \in a^*\},\ (j-i) \,\Box\, n\}$
consecutive_div(a, m, n)	Every a appears mod(m, n) times consecutively. / $\{w \mid \forall i \in \text{ind}(w, a) \text{ s.t. } w[i-1] \neq a \text{ and}$ $j = \arg\max_j \{w[i:j] \in a^*\},\ (j-i) \in \text{mod}(m, n)\}$

Table 3. Examples of incorrect regexes for 'Even number of a's', which has a possible solution $(b + ab^*a)^*$.

Error Type	**Regex**	**Error Analysis**
Syntactic error	$b + ab^*a$	Star operator is missing.
Logical error	$(a + ba^*b)^*$	Accepts "Even number of b's".
Semantic error	$(b^*ab^*ab^*)^*$	Does not accept strings consisting only of b's.

the provided logic formulas in Table 2. We aim at grading the submitted regex in terms of two types of syntactic correctness and a set of counterexamples as follows:

Syntactic grading Recall that previous approaches to computing the syntactic similarity or dissimilarity between two regexes rely on string edit-distance between two regexes. However, the string edit-distance between two regexes does not take the structural similarity into account. We instead use the *tree edit-distance* between two parse trees of regexes as the tree edit-distance better reflects the structural similarity of regexes. One of the advantages of using the tree edit-distance is that we can also easily identify semantically equivalent regexes when they are viewed as parse trees rather than as strings.

Then, we define the syntactic grade of R based on the minimum tree edit-distance between R and an unknown regex $\tilde{R}$ such that $L(\tilde{R}) = L(\phi)$. Formally speaking, the syntactic grade of R is defined as follows:

$$G_{\text{syn}} = G_{\text{full}} - w_{\text{syn}}(R) \cdot \min\{\text{ed}_{\text{rt}}(R, \tilde{R}) \mid L(\tilde{R}) = L(\phi)\}, \tag{1}$$

where G_{full} means the full grade (10 in our implementation). The function w_{syn} scales the deduct points based on the length of the submitted regex R because if R is very long and it requires a single edit, then we may consider that R is syntactically similar enough to a solution.

Let us explain the detailed procedure for computing G_{syn}. We first parse the regex R as a binary tree and construct the set $S_{R,n} = \{\tilde{R} \mid \text{ed}_{\text{rt}}(R, \tilde{R}) \leq n\}$ of regexes where each regex is within the tree edit-distance n ($n = 2$ in our experiments). Note that we use tree edit-distance instead of string edit-distance used in AT v3 and RegED as the tree edit-distance makes more sense to compute the syntactic difference between two regexes. For instance, the tree edit-distance between $a + b$ and $(b + a)^*$ is one while the string edit-distance is five.

For running the above procedure more efficiently, we increment the value of n from zero by one at each iteration until we find such $\tilde{R}$. We also check whether or not the current regex is already examined in the previous iteration by comparing the parse trees of regexes so that our implementation can avoid redundant regex equivalence tests.

Logical grading Given a problem ‘A regex for strings where the string *aba* appears at 3th position.’, a student may submit an incorrect solution $(a + b)aba(a + b)^*$ by making a mistake of reading the number ‘3’ as ‘2’. Because the most plausible answer is $(a+b)(a+b)aba(a+b)^*$, the student’s submission is likely to receive no partial grade according to the syntactic grading, which could be a harsh decision for an elementary mistake. However, if we semantically compare the submission and the problem, there is a hope to receive a partial grade as they turn out to be very similar in terms of corresponding logic formulas $\text{pos}(aba, 2)$ and $\text{pos}(aba, 3)$.

The main challenge in logical grading is to find a logic formula that corresponds to the submitted regex such that we can effectively quantify the amount of semantic discrepancy between the submitted regex and the problem. Given a regex, it requires a considerable amount of computation for finding a logic formula described as a logical combination of formulas provided in Table 2, assuming that the only feasible approach is an exhaustive tree search. Even worse, it is not always possible to find such a corresponding logic as the provided set of logic formulas cannot cover the entire class of regular languages. In order to save computation time, we instead utilize the solution logic formula by applying tree-level edits to the parse tree of the solution logic formula at most n times (again, $n = 2$ in our implementation) and checking whether the edited formula is language-equivalent to the submitted regex.

If we manage to find a logic formula $\tilde{\phi}$ that corresponds to the submitted regex, then the logical grade of R is then computed as follows:

$$G_{\log} = G_{\text{full}} - w_{\log}(\phi) \cdot \min\{\text{ed}_{\text{lt}}(\phi, \tilde{\phi}) \mid L(\tilde{\phi}) = L(R)\}. \tag{2}$$

Corner case grading In some cases, the submitted regex may describe a very similar language to the language in question although the regex is syntactically different (e.g., tree edit-distance is larger than n). For instance, let us consider a problem with the following description: "Strings with even number of a's." provided in Table 3. The language described by a regex $(b^*ab^*ab^*)^*$ is quite similar to the described language except for strings only with b's. In order to check whether the submitted regex deserves a corner case partial grade, we construct two DFAs for the following languages: $L(R) \cap \overline{L(\phi)}$ and $\overline{L(R)} \cap L(\phi)$. The language $L(R) \cap \overline{L(\phi)}$ is the set of strings that can be described by R and not by ϕ (false positive examples). On the contrary, $\overline{L(R)} \cap L(\phi)$ captures the set of strings that are described by ϕ but not by R (false negative examples). We enumerate the strings from both DFAs by using the `enumDFA` function in FAdo library in lexicographical order and display them to users to make them understand why their submissions are not correct by counterexamples.

We also assign a corner case grade $G_{\text{cor}} = \frac{4}{5} \times G_{\text{full}}$ if false positive and false negative sets satisfy one of the following conditions::

1. There is only ϵ in either false positive or negative set.
2. There are only less than m false positive and negative strings.
3. $L(R) \cup \{a^*\} = L(\phi)$ or $L(R) \cup \{b^*\} = L(\phi)$.

4.2 State Complexity of Logic Formula's DFAs

It is easy to see that all atomic logic formulas presented in Table 2 can be represented by DFAs of size linear in the lengths of string parameters. In the following proof, $m, n \in \mathbb{N}$, $a, b \in \Sigma$, $x, y \in \Sigma^*$, and $\Box \in \{>, =, <\}$.

Proposition 1. *For each atomic logic formula ϕ in Table 2, we can construct a DFA recognizing $L(\phi)$ with a polynomial number of states in $|x|$ and $|y|$.*

While most of the formulas in Table 2 can be represented as DFAs of size linear in the numerical parameters m and n as well, there are two exceptions: 'pos_rev(x, n)' and 'pos_every_rev(x, m, n)'.

Proposition 2. *For each atomic logic formula ϕ in Table 2 except* pos_rev(x, n) *and* pos_every_rev(x, m, n)*, we can construct a DFA recognizing $L(\phi)$ with a polynomial number of states in m and n.*

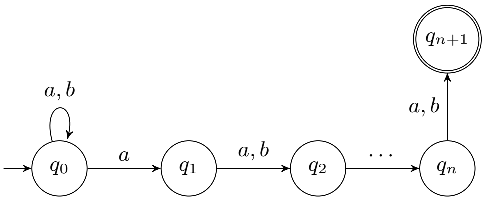

$$\begin{tikzcd} \arrow[r] & q_0 \arrow[loop above, "{a,b}"] \arrow[r, "a"] & q_1 \arrow[r, "{a,b}"] & q_2 \arrow[r, "\cdots"] & q_n \arrow[u, "{a,b}"] \\ & & & & q_{n+1} \end{tikzcd}$$

Fig. 2. An NFA for pos_rev(a, n).

Unlike the other formulas, the state complexity of pos_rev(x, n) and pos_every_rev(x, m, n) is exponential in n in the worst case.

Lemma 1. *The state complexity of* pos_rev(x, n) *is exponential in* n.

Proof. Since the NFA construction for pos_rev(x, n) requires $|x| + n + 1$ states, we have a simple upper bound $2^{|x|+n+1}$ which is exponential in n for the state complexity of pos_rev(x, n).

The simplest example where the lower bound is also exponential in n is when x is a string of length one such as a or b. See Fig. 2 for an NFA accepting the regular language pos_reverse(a, n). Since the initial state q_0 has a self-loop labeled by Σ, it is easy to see that the upper bound of the state complexity is 2^n as q_0 is always in the state set in the subset construction.

Now we will show that the upper bound 2^n can be reached by describing how we can reach any subset of states from $2^{\{q_1,q_2,\ldots,q_{n+1}\}}$. Let us consider a state set $P = \{q_{s_1}, q_{s_2}, \ldots, q_{s_k}\}$, where $s_i < s_j$ for $1 \le i < j \le k \le n + 1$. Then, we can reach P by reading the following string:

$$ab^{s_k - s_{k-1} - 1} ab^{s_{k-1} - s_{k-2} - 1} \cdots ab^{s_1 - 1}.$$

Since it is easy to see that all states in $2^{\{q_1,q_2,\ldots,q_{n+1}\}}$ are pairwise distinguishable, we conclude that the state complexity of pos_rev(a, n) is 2^n.

Now the following state complexity is obvious from the above observation.

Proposition 3. *The state complexity of* pos_every_rev(x, m, n) *is exponential in* n.

4.3 Heuristics for Faster Computation

In order to avoid this exponential blow-up in the size of DFAs, we employ the following two heuristics for faster computation of grades.

Regex reverse trick Interestingly, we can avoid this exponential blow-up caused by pos_rev(x, n) by reversing the given regex and the logic formula at the same time. We can trivially reverse the regex while maintaining the length and construct polynomial-sized DFAs for all reversed logic formulas except pos(x, n).

For instance, suppose that we are given a regex R and a declarative logic formula ϕ as follows:

$$R = a(a+b)b^*b \text{ and}$$
$$\phi = \text{pos\_rev}(b,n) \wedge \text{len}(>,3) \wedge \text{num}(a,>,1).$$

In order to avoid the exponential blow-up by $\text{pos\_rev}(x,n)$, we reverse R and ϕ as follows:

$$R' = bb^*(a+b)a \text{ and}$$
$$\phi' = \text{pos}(b,n) \wedge \text{len}(>,3) \wedge \text{num}(a,>,1).$$

Note that the logic such as $\text{len}(\Box,n)$ and $\text{len}(x,\Box,n)$ are reversal-invariant.

Concise Normal Form Recall that we construct a set of regexes from a submitted regex R by applying parse tree level edits for computing the syntactic grade. The main computational bottleneck comes from the repetitive regex equivalence tests as there are too many regexes in the set. In order to reduce the size of the constructed set, we employ the *concise normal form* [11] of regexes which are proven to be useful to sufficiently reduce the number of redundant regexes. For instance, we inductively apply substitution rules for subregexes such as $R^*R \to RR^*$, $R^*R^* \to R^*$, $R + R^* \to R^*$, $(R^*)^* \to R^*$ for concise regex representation and pruning of redundant regexes.

4.4 Description of Regex Grading Algorithm

Algorithm 1 precisely describes the whole procedure for computing the final grade of a student's regex R for a problem corresponding to a declarative logic formula ϕ. First, we preprocess the given student's regex R and declarative logic formula using the normal form and reverse trick for faster computation and convert them into the DFAs for partial grading. If the submission is equivalent to the solution, then give 10 points. If not, give the highest point among the three partial grades.

4.5 Converting Regex to NL Description

Many researchers have studied the problem of translating an NL description into a corresponding regex [13,15,17]. Here we examine a dual problem, namely, the problem of converting a regex into an NL description (Regex2NL) to help regex learners easily understand the language accepted by the given regex. Consider $(b+ab^*a)^*$ for an example again. Instead of merely translating the semantics of regex operators and symbols, our goal is to generate an 'easy-to-understand' NL description such as 'even number of a's' which corresponds to a logic formula defined in Table 1.

Our approach involves two steps, where we first find a logic formula corresponding to the regex and then translate the logic formula into an NL description

Algorithm 1: Our Regex Grading Algorithm

```
Input  : A student's regex $R$ and a declarative logic formula $\phi$
Output : A grade, feedback of $R$ for the problem specified by $\phi$, and a set of
         counter-examples
Convert $R$ into $R'$ which is in a regex normal form;
if $\phi$ contains pos_reverse$(x, n)$ and not pos$(x, n)$ then
    Reverse $R'$ and $\phi$;
Construct a DFA $A_{R'}$ for $R'$ and a DFA $A_\phi$ for $\phi$;
if $L(A_{R'}) = L(A_\phi)$ then
    if $|R'| < |R|$ then
        return 10, '$R$ can be written in more compact form such as $R'$', $\emptyset$;
    else
        return 10, 'Well constructed', $\emptyset$
else
    Compute $(G_{\text{syn}}, \tilde{R})$ and $(G_{\text{log}}, \tilde{\phi})$ of $R$;
    Generate a set $S$ of random strings from $L(\phi) \cap L(R)^c$;
    if $G_{\text{syn}} > G_{\text{log}}$ then
        return $G_{\text{syn}}$, '$R$ should include ... to be the $\tilde{R}$', $S$;
    else
        return $G_{\text{log}}$, '$R$ accepts a language specified by $\tilde{\phi}$', $S$;
```

by rules. It is worth noting again that there are regexes that cannot be effectively described by our logic. Therefore, it is not always possible to find a corresponding logic from a given regex even if we enumerate all logic formulas. Even if there exists a corresponding logic for the given regex, it takes too much time (more than one minute in general) for practical use in most cases. Hence we propose to use a deep learning-based approach that can predict a logic formula from a given regex with reasonably high accuracy in practical runtime (less than one second).

First, we train the Regex2Logic model that translates a regex to a logic formula using a sequence-to-sequence neural network with attention mechanism [3]. For training our Regex2Logic model, we use a dataset consisting of 13,437 pairs of regexes and logic formulas that are collected by time-consuming enumerations of regexes and logic formulas, and regex templates. We construct the regex-logic pair dataset for training our Regex2NL model which translates a given regex into a logic formula defined by using our simple declarative logic formulas. We collect the pairs by time-consuming enumerations of regexes and logic formulas and regex templates. We split the pairs into the ratio of 8:1:1 for training, validation, and test sets. We explain each process in more detail as follows:

1. Regex enumeration: enumerate regexes from the simplest one to more complex ones by increasing the depth of parse trees of regexes and searching for corresponding logic formulas until pre-defined thresholds (two for the depth, three for the length of argument strings and integers) for the complexity of logic formulas are reached.

Table 4. Statistics of the constructed regex-logic pair dataset used to train our Regex2NL model. ϕ, ϕ_1, and ϕ_2 denote atomic logic formulas found by enumerations of regexes and logic formulas or regex templates.

Logic Formula	**# Examples**
$\phi_1 \wedge \phi_2$	1,202
$\phi_1 \vee \phi_2$	1,939
$\neg\phi$	463
single_word(x)	88
pos(x, n)	3,854
pos_rev(x, n)	3,824
len$(\square, n)$	73
len_div(m, n)	20
pos_every(x, m, n)	0
num$(x, \square, n)$	954
num_div(x, m, n)	8
allX_followedbyY(x, y)	699
allX_followingY(x, y)	0
allX_beforeY(x, y, n)	184
exists_between(a, b, n)	0
consecutive$(a, \square, n)$	59
consecutive_div(a, m, n)	70
Total Number of Formulas	13,437

2. Logic formula enumeration: enumerate atomic logic formulas by varying the arguments such as strings of length up to n and integers from 1 to n and find a corresponding regex by exhaustively enumerating regexes.
3. Regex template: use regex templates for which we can easily match corresponding logic formulas. For instance, regexes with no operator such as aba correspond to the logic single_word(aba).

Table 4 shows the statistics of our dataset, especially in terms of the distribution of logic formulas used. The conjunction or disjunction of the same logic formulas is counted as a conjunction or disjunction.

In order to construct a set of regex-logic pairs, we can manually define a regex in a generalized form for each logic formula with arbitrary arguments. We rely on the following list of regex templates for generating various regexes by changing arguments of the templates:

- pos$(x, n) : \sigma^{(n-1)} x \sigma^*$
- pos_rev$(x, n) : \sigma^* x^R \sigma^{(n-1)}$
- len$(=, n) : \sigma^n$
- len$(<, n) : (\sigma + \lambda)^{n-1}$
- len$(<, n) : \sigma^? + \sigma^2 + \sigma^3 + ... + \sigma^{n-1}$
- len$(>, n) : \sigma^{n+1} \sigma^*$
- len_div$(x, m, n) : \sigma^n (\sigma^m)^*$
- len_div$(x, m, n) : (\sigma^m)^* \sigma^n$

By applying enumerated strings and integers as arguments, we can collect many regex-logic pairs. Once we discover the initial set of regex-logic pairs, we augment the data by combining the regexes and logic formulas with a regex operator $+$ and a logical connective $\vee$, respectively.

Note that our Regex2NL achieves about 92.3% prediction accuracy for the test set. For 167 incorrect regex submissions from students, our logical grading module finds 21 logic formulas that are within logic tree edit-distance two from the solution logic formula. Among the remaining 146 regexes, our model predicts 39 logic formulas that actually correspond to given regexes. We can provide natural language descriptions for 35.9% of the incorrect submissions from the logical grading module and the Regex2Logic model. We believe it is very useful to provide 'easy-to-understand' NL descriptions on 35.9% of submissions using our Regex2NL model, while most regexes do not have corresponding logic formulas definable by the proposed set of simple declarative logic formulas as we already discussed.

Then, we can transform the logic formula given by Regex2Logic to the natural language description with the heuristic template. We can make a template easily, as the logic formula has the characteristic of the natural language. We can use the entire framework of Regex2NL not only for feedback on incorrect submissions but also for making the random regex problem. For example, we can make the random regex first with regex enumeration of the regex template, then we can translate the regex to the natural language description. We can make the pair of regex-NL for using the regex problem.

4.6 Feedback Generation

There are natural types of feedback such as binary feedback (correct/wrong), an example, and a natural language-based conceptual hint. Binary feedback is the simplest yet necessary feedback that should be provided to students who submitted regexes. We can also simply generate a counterexample if the submitted regex is not correct. We focus on generating a natural language-based conceptual hint that describes the discrepancy between the desired solution and the submitted solution in an easily understandable manner.

When the submitted regex is not correct, there can be two cases as follows. First, the submitted regex should be slightly revised in order to accept the desired language. In this case, the most desirable feedback may be the way to revise the submitted regex. Second, the submitted regex accepts a semantically different language than the desired language as the student may have misinterpreted the question. Then, we may need to inform the student about the semantic discrepancy between the language described by the submitted regex and the desired regular language in an easily understandable manner.

For the first case, we provide the regex edit sequence between the submitted regex R and a regex R' which is syntactically closest (with the smallest regex edit-distance) to R while accepting the regular language specified in the problem. For the second case, we suggest the logic edit sequence between the logic formula ϕ corresponding to R and a logic formula $\tilde{\phi}$ specified in the problem. If the problem asks a regular language "strings containing a substring *abab* at least once" which corresponds to $\mathrm{num}(abab, >, 0)$ and the submitted regex captures a regular language corresponding to $\mathrm{num}(ab, =, 0)$, then we provide the following feedback: "Consider substring *abab* instead of *ab* and operator $>$ instead of $=$."

4.7 Converting Logic Formulas to NL Descriptions

Table 5 shows the NL descriptions for each atomic logic formula used in the rule-based translation of logic formulas into NL descriptions. When a logic formula is formed by combining more than two atomic formulas ϕ_1 and ϕ_2 using logical connectives, we simply combine the corresponding NL descriptions. For example, let $\mathrm{NL}(\phi)$ be the NL description of an atomic logic formula ϕ following the rules in Table 5. Then, $\mathrm{NL}(\phi_1 \wedge \phi_2)$ is defined as 'The set of strings that satisfy the following conditions: '$\mathrm{NL}(\phi_1)$' and '$\mathrm{NL}(\phi_2)$'.

Using this, we present regexes in more concise form even when the submitted regex is correct. Let us consider the problem 'all runs of a's have lengths that are multiples of three'. Note that a regex $(aaa + b)^*$ can be a solution. If a student submits $(aaa + b^*)^* + b^*$ as a solution, then the system should give the full grade since the submitted regex recognizes the desired regular language. While assigning a full grade to the submission, our algorithm provides $(aaa + b)^*$ to the student by computing the concise normal form [11] of the submission so that the student can recognize that there is a better solution (in terms of syntactic conciseness).

5 Experiments

We recruited 20 undergraduate students who were taking or had taken an automata course at the time of conducting our research, and ran our automatic grading algorithm on students' regex submissions for ten selected exercises from famous automata textbooks [9,14,18]. In order to compare the results of automated grading with the previous approaches including RegED [10] and AT v3 [5], we implemented the algorithms in Python 3 on our own and used them for comparison. We cannot use the existing implementations directly, because they do not support a feature of adjusting the maximum number of allowed edits, and not all of them are supported as a tool. We utilized the Python 3 port[4] of the `FAdo` [1] package, which is an open-source library for the symbolic manipulation of automata and other computation models. We also restricted the number of edits allowed for partial grades to two in our algorithm and AT v3, and one in RegED since RegED applies edits from both solutions and submissions.

5.1 Main Results

Table 6 shows the experimental results in terms of the statistics of grading results. We present the ratio of submissions that received partial grades by the considered

[4] https://github.com/0xnurl/fado-python3

Table 5. Natural language descriptions of our declarative logic formulas.

Logic Formula	**Description**
single_word(x)	only a single string x
pos(x, n)	strings that have substring x at nth position
pos_rev(x, n)	strings that have substring x at nth position in reverse order
len($=, n$)	strings of length n
len($<, n$)	strings shorter than n
len($>, n$)	strings longer than n
len_div($2, 0$)	strings of even-length
len_div($2, 1$)	strings of odd-length
len_div(m, n)	strings that have a remainder of n when it's length is divided by m
pos_every($x, 2, 0$)	strings in which character x appears every even-position
pos_every($x, 2, 1$)	strings in which character x appears every odd-position
pos_every(x, m, n)	strings in which substring x appears every kth position such that $k \mod m = n$
pos_every_rev($x, 2, 0$)	strings in which character x appears every even-position in reverse order
pos_every_rev($x, 2, 1$)	strings in which character x appears every odd-position in reverse order
pos_every_rev(x, m, n)	strings in which substring x appears every kth position in reverse order such that $k \mod m = n$
num($x, =, n$)	strings that contain x as a substring n times
num($x, <, n$)	strings that contain x as a substring less than n times
num($x, >, n$)	strings that contain x as a substring more than n times
num_div($x, 2, 0$)	strings that contain an even number of x's
num_div($x, 2, 1$)	strings that contain an odd number of x's
num_div(x, m, n)	strings that contain x's such that the number of its appearance modulo m is n
allX_followedbyY(x, y)	strings in which every substring x is followed by y
allX_followingY(x, y)	strings in which every substring x follows y
allX_beforeY(x, y)	strings in which every substring x appears before y
exists_between(x, y, n)	strings in which substring x appears n times between every pair of y
consecutive($x, =, n$)	strings in which every x appears n times consecutively
consecutive($x, <, n$)	strings in which every x appears less than n times consecutively
consecutive($x, >, n$)	strings in which every x appears more than n times consecutively
consecutive_div($x, 2, 0$)	strings in which every consecutive x's have even-length
consecutive_div($x, 2, 1$)	strings in which every consecutive x's have odd-length
consecutive_div(x, m, n)	strings in which every consecutive x's have a length such that when the length is divided by m, the remainder is n

grading algorithms in 'Partial Total' column. The 'Partial G_{syn}' column shows the ratio of regexes that received a partial 'syntactic grade' by AT v3, RegEd, and

Table 6. Performance comparisons of the proposed grading algorithm with baseline algorithms proposed in previous works [5,10].

Algorithm	**Partial** G_{syn}	**Partial** $G_{\log}$	**Partial Total**
AT v3 [5]	30.2%	7.0%	30.2%
RegED [10]	**45.3%**	9.3%	**45.3%**
Syntactic grading (Ours)	37.2%	8.7%	37.2%
Logical grading (Ours)	10.5%	**12.2%**	12.2%
Corner case grading (Ours)	6.4%	0.0%	6.4%
Our algorithm	39.0%	**12.2%**	40.7%

our syntactic grading algorithms over all regexes. Since AT v3 and RegED only consider syntactic grading, values in this column show the ratio of regexes that received partial grades over all regexes. On the other hand, 'Partial $G_{\log}$' column shows the ratio of regexes that received a partial 'logical grade' by our algorithm over all regexes. It is seen that AT v3 and RegED fail to assign partial grades to some regexes as they only consider syntactic differences with solution regexes, not the logic formulas behind the problem descriptions. Note that higher partial grades do not always mean that the grades are 'well-deserved'. It is important whether the partial grade is convincing. We will explain in the following section why RegED gives more partial grades than ours and why giving more partial grades cannot be a good choice.

To put it briefly, RegED gives partial grades to more regexes (45.3%) than AT v3 (30.2%) and even ours (40.7%). Table 7 shows several examples of the grades and feedback examples for students' submissions to the five problems in Table 1.

5.2 Validity of Grading Results

In order to verify that our algorithm indeed assigns partial grades to submissions that are 'well-deserved', we provide several reasons.

First, we can find logical partial grades while AT v3 and RegED cannot. We demonstrate two examples for the case. For the problem with the following description 'even number of a's', our algorithm assigns a partial grade to the submission $(a + ba^*b)^*$ while there is a possible solution $(b + ab^*a)^*$. Our logical grading module gives a partial grade, as it is possible that the student makes a simple mistake of confusing a with b. For the problem 'contains at most three a's', our algorithm assigns a partial grade to $b^*(a+\lambda)b^*(a+\lambda)b^*(a+\lambda)b^*(a+\lambda)b^*$ while one of the possible solutions is $b^*(a+\lambda)b^*(a+\lambda)b^*(a+\lambda)b^*$. This is again possible due to our logical grading module, as the student could have confused numbers.

Second, our syntactic grading gives some partial grades with tree-edit while others cannot. For example, our syntactic grading gives a partial grade to $(b^*a^*)abab(b^*a^*)$ for the problem 'contains the substring $abab$' as we may insert two star operators for the occurrences of (b^*a^*). However, RegED and AT

Table 7. Grading and feedback examples generated by our regex grading algorithm for problems in Table 1. We denote $a + b$ by σ for brevity.

No.	Student's Regex	Grade	Feedback Example
3	$\sigma^*(abab)^*\sigma^*$	7	Remove the star operator to convert it into $\sigma^*abab\sigma^*$.
3	$b^*(abab)^*a^*$	0	Should accept $\{aabab, ababb, aaabab, aababa\}$.
4	$a\sigma^*b$	3	Strings should begin with b instead of a (pos). Strings should end with a instead of b (pos_reverse).
4	$b\sigma^*a^*$	6	Insert a to convert it into $b\sigma^*a^*a$.
8	$(b^*ab^*ab^*)^*$	6	Include strings of b^* by inserting a union operator and b.
8	$(a + ba^*b)^*$	6	Strings should contain an even number of a's instead of b's (num_div).
11	$((b + \lambda)a)^*$	3	Each a instead of b should be followed by b instead of a (allX_followedbyY).
11	$(ab)^*$	3	Insert a union operator and λ to convert it into $((a + \lambda)b)^*$.
26	$(a + ba)^*bbb^*(a + ab)^*$	0	Should not accept $\{bbb, bbbb, abbba\}$.
26	$(a+ba)^*(b+\lambda)(a+ab)^*$	6	Insert a concatenation operator with b to convert it into $(a + ba)^*(bb + \lambda)(a + ab)^*$.

v3 will not assign a partial grade if they are provided $(a + b)^*abab(a + b)^*$ and $(b + a)^*abab(b + a)^*$ as possible solutions while our algorithm uses logic as a solution. This is because RegED utilizes only one solution regex for comparing with the submitted regex and it allows edits from both the solution and the submitted regex. RegED performs an edit at solution regex and submitted regex, respectively, to improve speed, but if solution regex is not given in an ideal form as in the above example, RegED cannot grade properly. To solve this problem, all possible variants of solution regex must be considered for editing and comparing and this leads to significant time-consuming. We can compare with every possible candidate without additional time, as our regex grading uses logic for the solution and permits the edit only in submission regex.

Third, the string edit used by RegED tends to cover too many candidates rather than our tree edit. For instance, it can change $a+b+c$ to a^*b+c and $aab+c$ with a single edit. This may differ depending on the TA's point of view, but we believe that the edit should be conducted more strictly due to the perspective of the tree structure, the original property of regex. Since given edits are more fluid than the tree edit, it allows more areas to be covered by edit, which is not considered the intended edit, suggesting that giving a lot of partial grading is not always the right direction. Assigning higher partial grades is not always the right direction, as it often jumps ahead of what we intended.

Table 8. Evaluation for the similarity with TA partial grades.

Algorithm	**Precision**	**Recall**	**F1 score**
AT v3 [5]	60.3%	50.8%	54.8%
RegED [10]	57.6%	**71.1**%	63.2%
Syntactic grading (Ours)	60.4%	53.3%	56.3%
Logical grading (Ours)	**73.3**%	25.8%	37.9%
Corner case grading (Ours)	60.0%	10.2%	17.3%
Our algorithm	62.2%	65.6%	**63.3**%

5.3 Comparison with TA Partial Grade

Table 8 demonstrates how the grading results by the algorithms align well with the human TAs' grading results. We ask five human TAs to give grades to 167 incorrect regex submissions by students. First, we calculate the precision, recall, and F1 score for each algorithm and for each TA. Precision is the percent of partial grades by the algorithm that matches the TA and recall is the percent of TA partial grades that the algorithm agrees with. Then we get an average score comparing the grading results with each result of human TAs. Since correct submissions should always receive full marks, we only consider incorrect submissions and check whether or not human TAs gave partial grades to the submissions. In other words, we assume that human TAs always make the right decisions in terms of giving partial grades to incorrect submissions and consider the cases where the partial grades are given as positive cases. We can see that the results in the 'Precision' column imply how the algorithms 'carefully' select submissions that deserve partial grades and the 'Recall' column show that the algorithms do not miss such cases.

Overall, our grading algorithm shows the best performance in terms of the F1 score, which is the harmonic mean of precision and recall. Then, RegED is places in the second position with a tiny gap between our algorithm and AT v3 following it.

Intuitively, it is natural that the recall is highest in RegED as RegED covers more regexes than the other compared algorithms. We can also see from the high precision of the logical grading module that the partial grade submissions captured by the logical grading module are quite precise even compared with the other modules used in our algorithm. However, the logical grading fails to capture the regexes that received partial grades by TAs from the other algorithms. On the other hand, the syntactic grading can capture much more regexes that received partial grades by TAs than the other modules in our algorithm. This also shows that human TAs tend to give partial grades to submissions with syntactic mistakes rather than to submissions with logical mistakes.

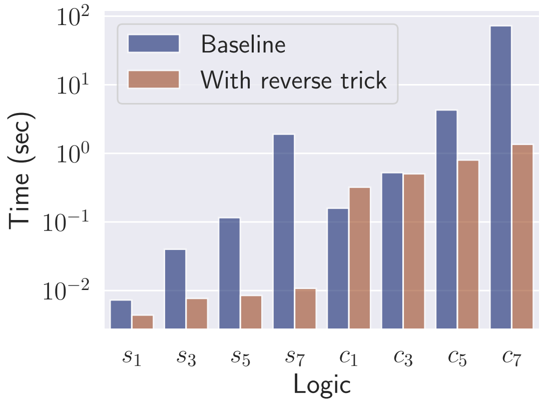


Fig. 3. Runtime comparison w/wo reverse trick. s_n and c_n indicate problems corresponding to logic formulas $\mathrm{pos\_rev}(a, n)$ and $\mathrm{pos\_rev}(a, n) \wedge \mathrm{num}(bba, >, 0)$, respectively.

5.4 Effectiveness of the Regex Reverse Trick

We demonstrate the effectiveness of the reverse trick in terms of runtime complexity reduction of the proposed algorithm in Fig. 3. There is no noticeable difference in short regexes. However, we can find that the time increases to log scale as the length of the regex increases.

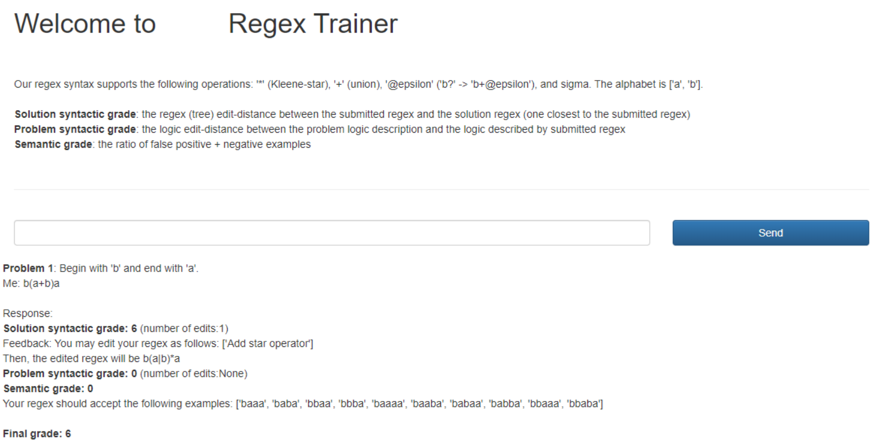
5.5 User Study

In Fig. 4, we provide a screenshot of a web page for the online ‘Regex Trainer’ in which our regex grading algorithm is employed. In the online Regex Trainer page, the system displays each regex construction problem in turn to a student. If the student inputs his/her answer for the problem, then the system shows the grade with feedback and displays the next problem.

We conducted a user study by asking five questions to nine students who performed tests on the usability and usefulness of our regex grading algorithm. The result is shown in Table 9. Each student is asked to give their answer to each question on a Likert scale from 1 (strongly disagree) to 5 (strongly agree). The result shows that average scores for the five questions are all in the range of [3.7, 4.4], which implies that the students in general find our grading system easy-to-use and useful for studying regexes.

5.6 Limitations

In the following, we leave a list of limitations of our study. First, the proposed set of logic formulas cannot express the entire class of regular languages. In future work, we may extend the set of formulas by adding useful logic formulas that



Welcome to Regex Trainer

Our regex syntax supports the following operations: '*' (Kleene-star), '+' (union), '@epsilon' ('b?' -> 'b+@epsilon'), and sigma. The alphabet is ['a', 'b'].

Solution syntactic grade: the regex (tree) edit-distance between the submitted regex and the solution regex (one closest to the submitted regex)
Problem syntactic grade: the logic edit-distance between the problem logic description and the logic described by submitted regex
Semantic grade: the ratio of false positive + negative examples

Problem 1: Begin with 'b' and end with 'a'.
Me: b(a+b)a

Response:
Solution syntactic grade: 6 (number of edits:1)
Feedback: You may edit your regex as follows: ['Add star operator']
Then, the edited regex will be b(a|b)*a
Problem syntactic grade: 0 (number of edits:None)
Semantic grade: 0
Your regex should accept the following examples: ['baaa', 'baba', 'bbaa', 'bbba', 'baaaa', 'baaba', 'babaa', 'babba', 'bbaaa', 'bbaba']

Final grade: 6

Fig. 4. A screenshot taken from the web page of online 'Regex Trainer' where our automatic grading module is used inside.

Table 9. Student survey result. Nine students gave their judgments for the following five questions on a Likert scale from 1 to 5.

Question	**Score**
The grading module is easy to use.	4.4
I agree with the given partial grade.	3.8
Feedback for each partial grade is helpful and instructive.	3.9
Feedback is not misleading.	3.7
Feedback and NL description improved my understanding of the regex.	3.9

are suitable for potential regex construction problems. Second, there could be another approach to catch student's 'mistakes'. We suggest three partial grades that catch syntactic, logical, and corner case mistakes. Finding a new cause of mistakes can provide richer and more detailed feedback for students. Moreover, it is very likely that our grading algorithm takes too much time if the submitted regex is unnecessarily long since in this case the number of regexes that should be examined would increase exponentially.

6 Conclusions

Due to the transition from face-to-face teaching to online, distance learning, the importance of developing an automated grading system has become more evident. We have presented an efficient and powerful automated grading algorithm for regexes in undergraduate automata and formal language courses. Our algorithm takes students' regex submissions and assigns appropriate grades with productive feedback to the regexes by considering the syntactic and semantic alignment between the submitted regexes and the problem definition. Moreover, by employing several heuristics such as the reverse trick and intermediate regex simplification, we could have reduced the runtime complexity for the repetitive regex equivalence tests for grading regexes.

Acknowledgments

We thank the reviewers for their valuable comments and suggestions for improving the presentation of the paper. This research was supported by the NRF grant (No. 2020R1A4A3079947), the IITP grant (No. 2022-0-00320), and the AI Graduate School Program (No. 2020-0-01361) funded by the Korea government (MSIT).

References

1. Almeida, A., Almeida, M., Alves, J., Moreira, N., Reis, R.: FAdo and GUItar. In: Proceedings of the 14th International Conference on Implementation and Application of Automata. pp. 65–74 (2009)
2. Alur, R., D'Antoni, L., Gulwani, S., Kini, D., Viswanathan, M.: Automated grading of DFA constructions. In: IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013. pp. 1976–1982. IJCAI/AAAI (2013)
3. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015)
4. Büchi, J.R.: Weak second-order arithmetic and finite automata. Journal of Symbolic Logic **28**(1) (1963)
5. D'Antoni, L., Helfrich, M., Kretínský, J., Ramneantu, E., Weininger, M.: Automata tutor v3. In: Proceedings of the 32nd International Conference on Computer Aided Verification. vol. 12225, pp. 3–14. Springer (2020)
6. D'Antoni, L., Kini, D., Alur, R., Gulwani, S., Viswanathan, M., Hartmann, B.: How can automatic feedback help students construct automata? ACM Transactions on Computer-Human Interaction **22**(2) (2015)
7. Davis, J.C., Michael IV, L.G., 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: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 443–454 (2019)
8. Henriksen, J.G., Jensen, J., Jørgensen, M., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 89–110 (1995)
9. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, MA, 2 edn. (1979)
10. Kakkar, H.: Automated Grading and Feedback of Regular Expressions. Master's thesis, Department of Computer Science (2017)

11. Kim, S.H., Im, H., Ko, S.K.: Efficient enumeration of regular expressions for faster regular expression synthesis. In: Implementation and Application of Automata. pp. 65–76. Springer International Publishing, Cham (2021)
12. Klarlund, N., Møller, A.: MONA Version 1.4 User Manual. BRICS, Department of Computer Science, Aarhus University (January 2001), notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3
13. Kushman, N., Barzilay, R.: Using semantic unification to generate regular expressions from natural language. In: NAACL-HLT '13. pp. 826–836 (2013)
14. Linz, P.: An Introduction to Formal Language and Automata. Jones and Bartlett Publishers, Inc., USA (2006)
15. Locascio, N., Narasimhan, K., DeLeon, E., Kushman, N., Barzilay, R.: Neural generation of regular expressions from natural language with minimal domain knowledge. In: Su, J., Carreras, X., Duh, K. (eds.) Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016. pp. 1918–1923. The Association for Computational Linguistics (2016)
16. 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)
17. Park, J., Ko, S., Cognetta, M., Han, Y.: Softregex: Generating regex from natural language descriptions using softened regex equivalence. In: EMNLP-IJCNLP '19. pp. 6424–6430 (2019)
18. Sipser, M.: Introduction to the Theory of Computation. Cengage Learning (2012)

Builtin Types Viewed as Inductive Families

Guillaume Allais(✉)

University of St Andrews, St Andrews, UK
guillaume.allais@ens-lyon.org

Abstract. State of the art optimisation passes for dependently typed languages can help erase the redundant information typical of invariant-rich data structures and programs. These automated processes do not dramatically change the *structure* of the data, even though more efficient representations could be available.
Using Quantitative Type Theory as implemented in Idris 2, we demonstrate how to define an invariant-rich, typechecking-time data structure packing an efficient runtime representation together with runtime irrelevant invariants. The compiler can then aggressively erase all such invariants during compilation.
Unlike other approaches, the complexity of the resulting representation is entirely predictable, we do not require both representations to have the same structure, and yet we are able to seamlessly program as if we were using the high-level structure.

Keywords: Quantitative Type Theory · Indexed families · Runtime representation · Idris 2

1 Introduction

Dependently typed languages have empowered users to precisely describe their domain of discourse by using inductive families [13]. Programmers can bake crucial invariants directly into their definitions thus refining both their functions' inputs and outputs. The constrained inputs allow them to only consider the relevant cases during pattern matching, while the refined outputs guarantee that client code can safely rely on the invariants being maintained. This programming style is dubbed 'correct by construction'.

However, relying on inductive families can have a non-negligible runtime cost if the host language is compiling them naïvely. And even state of the art optimisation passes for dependently typed languages cannot make miracles: if the source code is not efficient, the executable will not be either.

A state of the art compiler will for instance successfully compile length-indexed lists to mere lists thus reducing the space complexity from quadratic to linear in the size of the list. But, confronted with a list of booleans whose length is statically known to be less than 64, it will fail to pack it into a single machine word thus spending linear space when constant would have sufficed.

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 113–139, 2023.
https://doi.org/10.1007/978-3-031-30044-8_5

In section 2, we will look at an optimisation example that highlights both the strengths and the limitations of the current state of the art when it comes to removing the runtime overheads potentially incurred by using inductive families.

In section 3 we will give a quick introduction to Quantitative Type Theory, the expressive language that grants programmers the ability to have both strong invariants and, reliably, a very efficient runtime representation.

In section 4 we will look at an inductive family that we use in a performance-critical way in the TypOS project [2] and whose compilation suffers from the limitations highlighted in section 2. Our current and unsatisfactory approach is to rely on the safe and convenient inductive family when experimenting in Agda and then replace it with an unsafe but vastly more efficient representation in our actual Haskell implementation.

Finally in section 5, we will study the actual implementation of our efficient and invariant-rich solution implemented in Idris 2. We will also demonstrate that we can recover almost all the conveniences of programming with inductive families thanks to smart constructors and views.

2 An Optimisation Example

The prototypical examples of the naïve compilation of inductive families being inefficient are probably the types of vectors (`Vect`) and finite numbers (`Fin`). Their interplay is demonstrated by the `lookup` function. Let us study this example and how successive optimisation passes can, in this instance, get rid of the overhead introduced by using indexed families over plain data.

A vector is a length-indexed list. The type `Vect` is parameterised by the type of values it stores and indexed over a natural number corresponding to its length. More concretely, its `Nil` constructor builds an empty vector of size `Z` (i.e. zero), and its `(::)` (pronounced 'cons') constructor combines a value of type `a` (the head) and a subvector of size `n` (the tail) to build a vector of size (`S n`) (i.e. successor of n).

```
data Vect : Nat -> Type -> Type where
  Nil : Vect Z a
  (::) : a -> Vect n a -> Vect (S n) a
```

The size `n` is not explicitly bound in the type of `(::)`. In Idris 2, this means that it is automatically generalised over in a prenex manner reminiscent of the handling of free type variables in languages in the ML family. This makes it an implicit argument of the constructor. Consequently, given that `Nat` is a type of *unary* natural numbers, a naïve runtime representation of a (`Vect n a`) would have a size quadratic in `n`. A smarter representation with perfect sharing would still represent quite an overhead as observed by Brady, McBride, and McKinna [6].

A finite number is a number known to be strictly smaller than a given natural number. The type `Fin` is indexed by said bound. Its `Z` constructor models `0` and is bound by any non-zero bound, and its `S` constructor takes a number bound by n and returns its successor, bound by (1 + n). A naïve compilation would here also lead to a runtime representation suffering from a quadratic blowup.

```
data Fin : Nat -> Type where
  Z : Fin (S n)
  S : Fin n -> Fin (S n)
```

This leads us to the definition of the lookup function. Provided a vector of size n and a finite number k bound by this same n, we can define a *total* function looking up the value stored at position k in the vector. It is guaranteed to return a value. Note that we do not need to consider the case of the empty vector in the pattern matching clauses as all of the return types of the Fin constructors force the index to be non-zero and, because the vector and the finite number talk about the same n, having an empty vector would automatically imply having a value of type (Fin 0) which is self-evidently impossible.

```
lookup : Vect n a -> Fin n -> a
lookup (x :: _) Z = x
lookup (_ :: xs) (S k) = lookup xs k
```

Thanks to our indexed family, we have gained the ability to define a function that cannot possibly fail, as well as the ability to only talk about the pattern matching clauses that make sense. This seemed to be at the cost of efficiency but luckily for us there has already been extensive work on erasure to automatically detect redundant data [6] or data that will not be used at runtime [22].

2.1 Optimising Vect, Fin, and lookup

An analysis in the style of Brady, McBride, and McKinna's [6] can solve the quadratic blowup highlighted above by observing that the natural number a vector is indexed by is entirely determined by the spine of the vector. In particular, the length of the tail does not need to be stored as part of the constructor: it can be reconstructed as the predecessor of the length of the overall vector. As a consequence, a vector can be adequately represented at runtime by a pair of a natural number and a list. Similarly a bounded number can be adequately represented by a pair of natural numbers. Putting all of this together and remembering that the vector and the finite number share the same n, lookup can be compiled to a function taking two natural numbers and a list. In Idris 2 we would write the optimised lookup as follows (we use the **partial** keyword because this transformed version is not total at that type).

```
partial
lookup : (n : Nat) -> List a -> Nat -> a
lookup (S n) (x :: _) Z = x
lookup (S n) (_ :: xs) (S k) = lookup n xs k
```

We can see in the second clause that the recursive call is performed on the tail of the list (formerly vector) and so the first argument to lookup corresponding to the vector's size is decreased by one. The invariant, despite not being explicit anymore, is maintained.

A Tejiščák-style analysis [22] can additionally notice that the lookup function does not use the bound's value and drop it. This leads to the lookup function on vectors being compiled to its partial-looking counterpart acting on lists.

```
partial
lookup : List a -> Nat -> a
lookup (x :: _) Z = x
lookup (_ :: xs) (S k) = lookup xs k
```

Even though this is in our opinion a pretty compelling example of erasing away the apparent complexity introduced by inductive families, this approach has two drawbacks.

Firstly, it relies on the fact that the compiler can and will automatically perform these optimisations. But nothing in the type system prevents users from inadvertently using a value they thought would get erased, thus preventing the Tejiščák-style optimisation from firing. In performance-critical settings, users may rather want to state their intent explicitly and be kept to their word by the compiler in exchange for predictable and guaranteed optimisations.

Secondly, this approach is intrinsically limited to transformations that preserve the type's overall structure: the runtime data structures are simpler but very similar still. We cannot expect much better than that. It is so far unrealistic to expect e.g. a change of representation to use a balanced binary tree instead of a list in order to get logarithmic lookups rather than linear ones.

2.2 No Magic Solution

Even if we are able to obtain a more compact representation of the inductive family at runtime through enough erasure, this does not guarantee runtime efficiency. As the Coq manual [11] reminds its users, extraction does not magically optimise away a user-defined quadratic multiplication algorithm when extracting unary natural numbers to an efficient machine representation. In a pragmatic move, Coq, Agda, and Idris 2 all have ad-hoc rules to replace convenient but inefficiently implemented numeric functions with asymptotically faster counterparts in the target language.

However this approach is not scalable: if we may be willing to extend our trusted core to a high quality library for unbounded integers, we do not want to replace our code only proven correct thanks to complex invariants with a wildly different untrusted counterpart purely for efficiency reasons.

In this paper we use Quantitative Type Theory [16,4] as implemented in Idris 2 [5] to bridge the gap between an invariant-rich but inefficient representation based on an inductive family and an unsafe but efficient implementation using low-level primitives. Inductive families allow us to *view* [24,18] the runtime relevant information encoded in the low-level and efficient representation as an information-rich compile time data structure. Moreover the quantity annotations guarantee the erasure of this additional information during compilation.

3 Some Key Features of Idris 2

Idris 2 implements Quantitative Type Theory, a Martin-Löf type theory enriched with a semiring of quantities classifying the ways in which values may be used. In a type, each binder is annotated with the quantity by which its argument must abide.

3.1 Quantities

A value may be *runtime irrelevant*, *linear*, or *unrestricted*.

Runtime irrelevant values (**0** quantity) cannot possibly influence control flow as they will be erased entirely during compilation. This forces the language to impose strong restrictions on pattern-matching over these values. Typical examples are types like the `a` parameter in (`List a`), or indices like the natural number `n` in (`Vect n a`). These are guaranteed to be erased at compile time. The advantage over a Tejiščák-style analysis is that users can state their intent that an argument ought to be runtime irrelevant and the language will insist that it needs to be convinced it indeed is.

Linear values (**1** quantity) have to be used exactly once. Typical examples include the `%World` token used by Idris 2 to implement the `IO` monad à la Haskell, or file handles that cannot be discarded without first explicitly closing the file. At runtime these values can be updated destructively. We will not use linearity in this paper.

Last, *unrestricted* values (denoted by no quantity annotation) can flow into any position, be duplicated or thrown away. They are the usual immutable values of functional programming.

The most basic of examples mobilising both the runtime irrelevance and unrestricted quantities is the identity function.

```
id : {0 a : Type} -> (x : a) -> a
id x = x
```

Its type starts with a binder using curly braces. This means it introduces an implicit variable that does not need to be filled in by the user at call sites and will be reconstructed by unification. The variable it introduces is named `a` and has type `Type`. It has the **0** quantity annotation which means that this argument is runtime irrelevant and so will be erased during compilation.

The second binder uses parentheses. It introduces an explicit variable whose name is `x` and whose type is the type `a` that was just bound. It has no quantity annotation which means it will be an unrestricted variable.

Finally the return type is the type `a` bound earlier. This is, as expected, a polymorphic function from `a` to `a`. It is implemented using a single clause that binds `x` on the left-hand side and immediately returns it on the right-hand side.

If we were to try to annotate the binder for `x` with a **0** quantity to make it runtime irrelevant then Idris 2 would rightfully reject the definition. The following **`failing`** block shows part of the error message complaining that `x` cannot be used at an unrestricted quantity on the right-hand side.

```
failing "x is not accessible in this context."
  id : {0 a : Type} -> (0 x : a) -> a
  id x = x
```

3.2 Proof Search

In Idris 2, Haskell-style ad-hoc polymorphism [25] is superseded by a more general proof search mechanism. Instead of having blessed notions of type classes, instances and constraints, the domain of any dependent function type can be marked as **auto**. This signals to the compiler that the corresponding argument will be an implicit argument and that it should not be reconstructed by unification alone but rather by proof search. The search algorithm will use the appropriate user-declared hints as well as the local variables in scope.

By default, a datatype's constructors are always added to the database of hints. And so the following declaration brings into scope both an indexed family `So` of proofs that a given boolean is `True`, and a unique constructor `Oh` that is automatically added as a hint.

```
data So : Bool -> Type where
  Oh : So True
```

As a consequence, we can for instance define a record type specifying what it means for `n` to be an even number by storing its `half` together with a proof that is both runtime irrelevant and filled in by proof search. Because (`2 * 3 == 6`) computes to `True`, Idris 2 is able to fill-in the missing proof in the definition of `even6` using the `Oh` hint.

```
record Even (n : Nat) where
  constructor MkEven
  half : Nat
  {auto 0 prf : So (2 * half == n)}
```

```
even6 : Even 6
even6 = MkEven { half = 3 }
```

We will use both `So` and the **auto** mechanism in section 5.3.

3.3 Application: `Vect`, as `List`

We can use the features of Quantitative Type Theory to give an implementation of `Vect` that is guaranteed to erase to a `List` at runtime independently of the optimisation passes implemented by the compiler. The advantage over the optimisation passes described in section 2 is that the user has control over the runtime representation and does not need to rely on these optimisations being deployed by the compiler.

The core idea is to make the slogan 'a vector is a length-indexed list' a reality by defining a record packing together the `encoding` as a list and a proof its length is equal to the expected `Nat` index. This proof is marked as runtime irrelevant to ensure that the list is the only thing remaining after compilation.

```
record Vect (n : Nat) (a : Type) where
  constructor MkVect
  encoding : List a
  0 valid : length encoding === n
```

Smart constructors Now that we have defined vectors, we can recover the usual building blocks for vectors by defining smart constructors, that is to say functions `Nil` and `(::)` that act as replacements for the inductive family's data constructors.

```
  Nil : Vect Z a
  Nil = MkVect [] Refl
```

The smart constructor `Nil` returns an empty vector. It is, unsurprisingly, encoded as the empty list (`[]`). Because (`length []`) statically computes to `Z`, the proof that the encoding is valid can be discharged by reflexivity.

```
  (::) : a -> Vect n a -> Vect (S n) a
  x :: MkVect xs eq = MkVect (x :: xs) (cong S eq)
```

Using `(::)` we can combine a head and a tail of size `n` to obtain a vector of size (`S n`). The encoding is obtained by consing the head in front of the tail's encoding and the proof this is valid (`cong S eq`) uses the fact that propositional equality is a congruence and that (`length` (`x :: xs`)) computes to (`S` (`length xs`)).

View Now that we know how to build vectors, we demonstrate that we can also take them apart using a view.

A view for a type T, in the sense of Wadler [24], and as refined by McBride and McKinna [18], is an inductive family V indexed by T together with a total function mapping every element t of T to a value of type $(V t)$. This simple gadget provides a powerful, user-extensible, generalisation of pattern-matching. Patterns are defined inductively as either a pattern variable, a forced term (i.e. an arbitrary expression that is determined by a constraint arising from another pattern), or a data constructor fully applied to subpatterns. In contrast, the return indices of an inductive family's constructors can be arbitrary expressions.

In the case that interests us, the view allows us to emulate 'matching' on which of the two smart constructors `Nil` or `(::)` was used to build the vector being taken apart.

```
data View : Vect n a -> Type where
  Nil : View Nil
  (::) : (x : a) -> (xs : Vect n a) -> View (x :: xs)
```

The inductive family `View` is indexed by a vector and has two constructors corresponding to the two smart constructors. We use Idris 2's overloading capabilities to give each of the `View`'s constructors the name of the smart constructor it corresponds to. By pattern-matching on a value of type (`View xs`), we will be able to break `xs` into its constitutive parts and either observe it is equal to `Nil` or recover its head and its tail.

```
  view : (xs : Vect n a) -> View xs
  view (MkVect [] Refl) = Nil
  view (MkVect (x :: xs) Refl) = x :: MkVect xs Refl
```

The function `view` demonstrates that we can always tell which constructor was used by inspecting the `encoding` list. If it is empty, the vector was built using the `Nil` smart constructor. If it is not then we got our hands on the head and the tail of the encoding and (modulo some re-wrapping of the tail) they are effectively the head and the tail that were combined using the smart constructor.

Application: `map` We can then use these constructs to implement the function `map` on vectors without ever having to explicitly manipulate the encoding. The maximally sugared version of `map` is as follows:

```
map : (a -> b) -> Vect n a -> Vect n b
map f xs@_ with (view xs)
  _ | [] = []
  _ | hd :: tl = f hd :: map f tl
```

On the left-hand side the view lets us seamlessly pattern-match on the input vector. Using the **`with`** keyword we have locally modified the function definition so that it takes an extra argument, here the result of the intermediate computation (`view xs`). Correspondingly, we have two clauses matching on this extra argument; the symbol `|` separates the original left-hand side (here elided using `_` because it is exactly the same as in the parent clause) from the additional pattern. This pattern can either have the shape `[]` or (`hd :: tl`) and, correspondingly, we learn that `xs` is either `[]` or (`hd :: tl`).

On the right-hand side the smart constructors let us build the output vector. Mapping a function over the empty vector yields the empty vector while mapping over a cons node yields a cons node whose head and tail have been modified.

This sugared version of `map` is equivalent to the following more explicit one:

```
map : (a -> b) -> Vect n a -> Vect n b
map f xs with (view xs)
  map f .([]) | [] = []
  map f .(hd :: tl) | hd :: tl = f hd :: map f tl
```

In the parent clause we have explicitly bound `xs` instead of merely introducing an alias for it by writing (`xs@_`) and so we will need to be explicit about the ways in which this pattern is refined in the two with-clauses.

In the with-clauses, we have explicitly repeated the refined version of the parent clause's left-hand side. In particular we have used dotted patterns to insist that `xs` is now entirely *forced* by the match on the result of (`view xs`).

We have seen that by matching on the result of the (`view xs`) call, we get to 'match' on `xs` as if `Vect` were an inductive type. This is the power of views.

Application: `lookup` The type (`Fin n`) can similarly be represented by a single natural number and a runtime irrelevant proof that it is bound by `n`. We leave these definitions out, and invite the curious reader to either attempt to implement them for themselves or look at the accompanying code.

Bringing these definitions together, we can define a `lookup` function which is similar to the one defined in section 2.

```
lookup : Vect n a -> Fin n -> a
lookup xs@_ k@_ with (view xs) | (view k)
  _ | hd :: _ | Z = hd
  _ | _ :: tl | S k' = lookup tl k'
```

We are seemingly using `view` at two different types (`Vect` and `Fin` respectively) but both occurrences actually refer to separate functions: Idris 2 lets us overload functions and performs type-directed disambiguation.

For pedagogical purposes, this sugared version of `lookup` can also be expanded to a more explicit one that demonstrates the views' power.

```
lookup : Vect n a -> Fin n -> a
lookup xs k with (view xs) | (view k)
  lookup .(hd :: tl) .(Z) | hd :: tl | Z = hd
  lookup .(hd :: tl) .(S k') | hd :: tl | S k' = lookup tl k'
```

The main advantage of this definition is that, based on its type alone, we know that this function is guaranteed to be processing a list and a single natural number at runtime. This efficient runtime representation does not rely on the assumption that state of the art optimisation passes will be deployed.

We have seen some of Idris 2's powerful features and how they can be leveraged to empower users to control the runtime representation of the inductive families they manipulate. This simple example only allowed us to reproduce the performance that could already be achieved by compilers deploying state of the art optimisation passes. In the following sections, we are going to see how we can use the same core ideas to compile an inductive family to a drastically different runtime representation while keeping good high-level ergonomics.

4 Thinnings, Cooked Two Ways

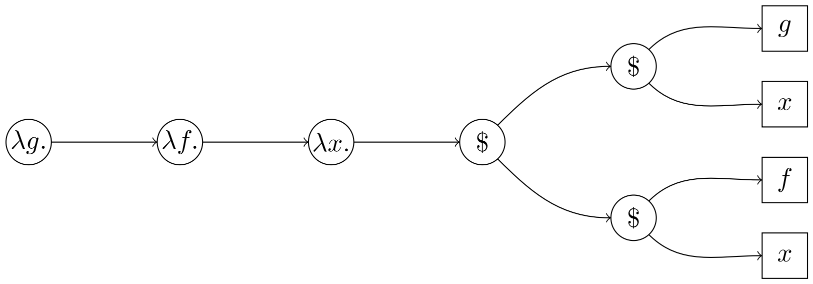
We experienced a major limitation of compilation of inductive families during our ongoing development of TypOS [2], a domain specific language to define concurrent typecheckers and elaborators. Core to this project is the definition of actors manipulating a generic notion of syntax with binding. Internally the terms of this syntax with binding are based on a co-de Bruijn representation (an encoding we will explain below) which relies heavily on thinnings. A thinning (also known as an Order Preserving Embedding [9]) between a source and a target scope is an order preserving injection of the smaller scope into the larger one. They are usually represented using an inductive family. The omnipresence of thinnings in the co-de Bruijn representation makes their runtime representation a performance critical matter.

Let us first remind the reader of the structure of abstract syntax trees in a named, a de Bruijn, and a co-de Bruijn representation. We will then discuss two representations of thinnings: a safe and convenient one as an inductive family, and an unsafe but efficient encoding as a pair of arbitrary precision integers.

4.1 Named, de Bruijn, and co-de Bruijn Syntaxes

In this section we will use the S combinator $(\lambda g.\lambda f.\lambda x.gx(fx))$ as a running example and represent terms using a syntax tree whose constructor nodes are circles and variable nodes are squares. To depict the S combinator we will only need λ-abstraction and application (rendered \$) nodes. A constructor's arguments become its children in the tree. The tree is laid out left-to-right and a constructor's arguments are displayed top-to-bottom.

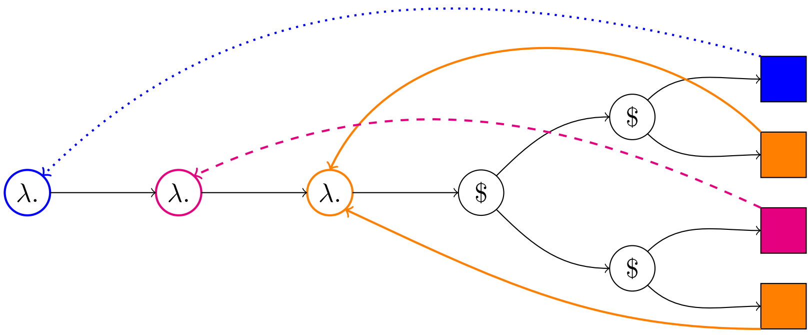
Named Syntax The first representation is using explicit names. Each binder has an associated name and each variable node carries a name. A variable refers to the closest enclosing binder which happens to be using the same name.



To check whether two terms are structurally equivalent (α-*equivalence*) potentially requires renaming bound names. In order to have a simple and cheap α-equivalence check we can instead opt for a nameless representation.

De Bruijn Syntax An abstract syntax tree based on de Bruijn indices [8] replaces names with natural numbers counting the number of binders separating a variable from its binding site. The S combinator is now written $(\lambda\,\lambda\,\lambda\,2\,0\,(1\,0))$.

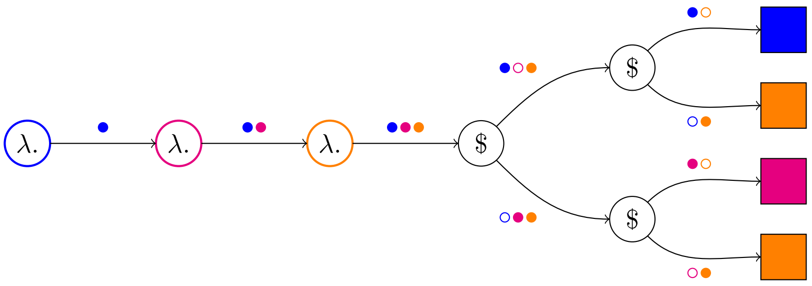
You can see in the following graphical depiction that λ-abstractions do not carry a name anymore and that variables are simply pointing to the binder that introduced them. We have left the squares empty but in practice the various coloured arrows would be represented by a natural number. For instance the dashed magenta one corresponds to 1 because you need to ignore one λ-abstraction (the orange one) on your way towards the root of the tree before you reach the corresponding magenta binder.



To check whether a subterm does not mention a given set of variables (a *thickening* test, the opposite of a *thinning* which extends the current scope with unused variables), you need to traverse the whole term. In order to have a simple cheap thickening test we can ensure that each subterms knows precisely what its *support* is and how it embeds in its parent's.

Co-de Bruijn Syntax In a co-de Bruijn representation [17] each subterm selects exactly the variables that stay in scope for that term, and so a variable constructor ultimately refers to the only variable still in scope by the time it is reached. This representation ensures that we know precisely what the scope of a given term currently is.

In the following graphical rendering, we represent thinnings as lists of full (●) or empty (○) discs depending on whether the corresponding variable is either kept or discarded. For instance the thinning represented by ○●● throws the blue variable away, and keeps both the magenta and orange ones.



We can see that in such a representation, each node in the tree stores one thinning per subterm. This will not be tractable unless we have an efficient representation of thinnings.

4.2 The Performance Challenges of co-de Bruijn

Using the co-de Bruijn approach, a term in an arbitrary context is represented by the pairing of a term in co-de Bruijn syntax with a thinning from its support into the wider scope. Having such a precise handle on each term's support allows us to make operations such as thinning, substitution, unification, or common sub-expression elimination more efficient.

Thinning a term does not require us to traverse it anymore. Indeed, embedding a term in a wider context will not change its support and so we can simply compose the two thinnings while keeping the term the same.

Substitution can avoid traversing subterms that will not be changed. Indeed, it can now easily detect when the substitution's domain does not intersect with the subterm's support.

Unification requires performing thickening tests when we want to solve a metavariable declared in a given context with a terms seemingly living in a wider one. We once more do not need to traverse the term to perform this test, and can simply check whether the outer thinning can be thickened.

Common sub-expression elimination requires us to identify alpha-equivalent terms potentially living in different contexts. Using a de Bruijn representation, these can be syntactically different: a variable represented by the natural number v in Γ would be $(1{+}v)$ in Γ, σ but $(2{+}v)$ in Γ, τ, ν. A co-de Bruijn representation, by discarding all the variables not in the support, guarantees that we can once more use syntactic equality to detect alpha-equivalence. This encoding is used for instance (albeit unknowingly) by Maziarz, Ellis, Lawrence, Fitzgibbon, and Peyton-Jones in their 'Hashing modulo alpha-equivalence' work [14].

For all of these reasons we have, as we mentioned earlier, opted for a co-de Bruijn representation in the implementation of TypOS [2]. And so it is crucial for performance that we have a compact representation of thinnings.

Thinnings in TypOS We first carefully worked out the trickier parts of the implementation in Agda before porting the resulting code to Haskell. This process highlighted a glaring gap between on the one hand the experiments done using a strongly typed inductive representation of thinnings and on the other hand their more efficient but unsafe encoding in Haskell.

Agda The Agda-based experiments use inductive families that make the key invariants explicit which helps tracking complex constraints and catches design flaws at typechecking time. The indices guarantee that we always transform the thinnings appropriately when we add or remove bound variables. In Idris 2, the inductive family representation of thinnings would be written:

```
data Thinning : (sx, sy : SnocList a) -> Type where
  Done : Thinning [<] [<]
  Keep : Thinning sx sy -> (0 x : a) -> Thinning (sx :< x) (sy :< x)
  Drop : Thinning sx sy -> (0 x : a) -> Thinning sx (sy :< x)
```

The `Thinning` family is indexed by two scopes (represented as snoclists i.e. lists that are extended from the right, just like contexts in inference rules): `sx` the tighter scope and `sy` the wider one. The `Done` constructor corresponds to a thinning from the empty scope to itself (`[<]` is Idris 2 syntactic sugar for the empty snoclist), and `Keep` and `Drop` respectively extend a given thinning by keeping or dropping the most local variable (`:<` is the 'snoc' constructor, a sort of flipped 'cons'). The 'name' (`x` of type `a`) is marked with the quantity **`0`** to ensure it is erased at compile time (cf. section 3).

During compilation, Idris 2 would erase the families' indices as they are forced (in the sense of Brady, McBride, and McKinna [6]), and drop the constructor arguments marked as runtime irrelevant. The resulting inductive type would be the following simple data type.

```
data Thinning = Done | Keep Thinning | Drop Thinning
```

At runtime this representation is therefore essentially a linked list of booleans (`Done` being `Nil`, and `Keep` and `Drop` respectively (`True ::`) and (`False ::`)).

Haskell The Haskell implementation uses this observation and picks a packed encoding of this list of booleans as a pair of integers. One integer represents the length `n` of the list, and the other integer's `n` least significant bits encode the list as a bit pattern where `1` is `Keep` and `0` is `Drop`.

Basic operations on thinnings are implemented by explicitly manipulating individual bits. It is not indexed and thus all the invariant tracking has to be done by hand. This has led to numerous and hard to diagnose bugs.

Thinnings in Idris 2 Idris 2 is a self-hosting language whose core datatype is currently based on a well-scoped de Bruijn representation. This precise indexing of terms by their scope helped entirely eliminate a whole class of bugs that plagued Idris 1's unification machinery.

If we were to switch to a co-de Bruijn representation for our core language we would want, and should be able, to have the best of both worlds: a safe *and* efficient representation!

Thankfully Idris 2 implements Quantitative Type Theory (QTT) which gives us a lot of control over what is to be runtime relevant and what is to be erased during compilation. This should allow us to insist on having a high-level interface that resembles an inductive family while ensuring that everything but a pair of integers is erased at compile time. We will exploit the key features of QTT presented in section 3 to have our cake and eat it.

5 An Efficient Invariant-Rich Representation

We can combine both approaches highlighted in section 4.2 by defining a record parameterised by a source (`sx`) and target (`sy`) scopes corresponding to the two ends of the thinnings, just like we would for the inductive family. This record packs two numbers and a runtime irrelevant proof.

Firstly, we have a natural number called `bigEnd` corresponding to the size of the big end of the thinning (`sy`). We are happy to use a (unary) natural number here because we know that Idris 2 will compile it to an unbounded integer.

Secondly, we have an integer called `encoding` corresponding to the thinning represented as a bit vector stating, for each variable, whether it is kept or dropped. We only care about the integer's bigEnd least significant bits and assume the rest is set to 0.

Thirdly, we have a runtime irrelevant proof invariant that encoding is indeed a valid encoding of size bigEnd of a thinning from sx to sy. We will explore the definition of the relation Invariant later on in section 5.3.

```
record Th {a : Type} (sx, sy : SnocList a) where
  constructor MkTh
  bigEnd : Nat
  encoding : Integer
  0 invariant : Invariant bigEnd encoding sx sy
```

The first sign that this definition is adequate is our ability to construct any valid thinning. We demonstrate it is the case by introducing functions that act as smart constructor analogues for the inductive family's data constructors.

5.1 Smart Constructors for Th

The first and simplest one is done, a function that packs a pair of 0 (the size of the big end, and the empty encoding) together with a proof that it is an adequate encoding of the thinning from the empty scope to itself. In this instance, the proof is simply the Done constructor.

```
done : Th [<] [<]
done = MkTh { bigEnd = 0, encoding = 0, invariant = Done }
```

To implement both keep and drop, we are going to need to perform bit-level manipulations. These are made easy by Idris 2's Bits interface which provides us with functions to shift the bit patterns left or right (shiftl, shiftr), set or clear bits at specified positions (setBit, clearBit), take bitwise logical operations like disjunction (.|.) or conjunction (.&.), etc.

In both keep and drop, we need to extend the encoding with an additional bit. For this purpose we introduce the cons function which takes a bit b and an existing encoding bs and returns the new encoding $bs \cdot b$.

```
cons : Bool -> Integer -> Integer
cons b bs = let bs0 = bs `shiftL` 1 in
            if b then (bs0 `setBit` 0) else bs0
```

No matter what the value of the new bit is, we start by shifting the encoding to the left to make space for it; this gives us bs0 which contains the bit pattern $bs \cdot 0$. If the bit is True then we need to additionally set the bit at position 0 to obtain $bs \cdot 1$. Otherwise if the bit is False, we can readily return the $bs \cdot 0$ encoding obtained by left shifting. The correctness of this function is backed by two lemma: testing the bit at index 0 after consing amounts to returning the cons'd bit, and shifting the cons'd encoding to the right takes us back to the unextended encoding.

```
testBit0Cons : (b : Bool) -> (bs : Integer) ->
               testBit (cons b bs) 0 === b

consShiftR : (b : Bool) -> (bs : Integer) ->
             (cons b bs) `shiftR` 1 === bs
```

The keep smart constructor demonstrates that from a thinning from sx to sy and a runtime irrelevant variable x we can compute a thinning from the extended source scope (sx :< x) to the target scope (sy :< x) where x was kept.

```
keep : Th sx sy -> (0 x : a) -> Th (sx :< x) (sy :< x)
keep th x = MkTh
  { bigEnd = S (th .bigEnd)
  , encoding = cons True (th .encoding)
  , invariant =
     let 0 b = eqToSo $ testBit0Cons True (th .encoding) in
     Keep (rewrite consShiftR True (th .encoding) in th.invariant) x
  }
```

The outer scope has grown by one variable and so we increment bigEnd. The encoding is obtained by cons-ing the boolean True to record the fact that this new variable is kept. Finally, we use the two lemmas shown above to convince Idris 2 the invariant has been maintained.

Similarly the drop function demonstrates that we can compute a thinning getting rid of the variable x freshly added to the target scope.

```
drop : Th sx sy -> (0 x : a) -> Th sx (sy :< x)
drop th x = MkTh
  { bigEnd = S (th .bigEnd)
  , encoding = cons False (th .encoding)
  , invariant =
    let 0 prf = testBit0Cons False (th .encoding)
        0 nb = eqToSo $ cong not prf in
    Drop (rewrite consShiftR False (th .encoding) in th .invariant) x
  }
```

We once again increment the bigEnd, use cons to record that the variable is being discarded and use the lemmas ensuring its correctness to convince Idris 2 the invariant is maintained.

We can already deploy these smart constructors to implement functions producing thinnings. We use which as our example. It is a filter-like function that returns a dependent pair containing the elements that satisfy a boolean predicate together with a proof that there is a thinning embedding them back into the input snoclist.

```
which : (a -> Bool) -> (sy : SnocList a) ->
        (sx : SnocList a ** Th sx sy)
which p [<] = ([<] ** done)
which p (sy :< y) =
  let (sx ** th) = which p sy in
  if p y then (sx :< y ** keep th y)
         else (sx ** drop th y)
```

If the input snoclist is empty then the output shall also be, and `done` builds a thinning from `[<]` to itself. If it is not empty we can perform a recursive call on the tail of the snoclist and then depending on whether the predicates holds true of the head we can either `keep` or `drop` it.

We are now equipped with these smart constructors that allow us to seamlessly build thinnings. To recover the full expressive power of the inductive family, we also need to be able to take these thinnings apart. Let us now tackle this issue.

5.2 Pattern Matching on `Th`

The `View` family is a sum type indexed by a thinning. It has one data constructor associated to each smart constructor and storing its arguments.

```
data View : Th sx sy -> Type where
  Done : View done
  Keep : (th : Th sx sy) -> (0 x : a) -> View (keep th x)
  Drop : (th : Th sx sy) -> (0 x : a) -> View (drop th x)
```

The accompanying `view` function witnesses the fact that any thinning arises as one of these three cases.

```
view : (th : Th sx sy) -> View th
```

We show the implementation of `view` in its entirety but leave out the technical auxiliary lemma it invokes. The interested reader can find them in the accompanying material. We will however inspect the code `view` compiles to after erasure in section 5.5 to confirm that these auxiliary definitions do not incur any additional runtime cost.

We first start by pattern matching on the `bigEnd` of the thinning. If it is `0` then we know the thinning has to be the empty thinning. Thanks to an inversion lemma called `isDone`, we can collect a lot of equality proofs: the encoding `bs` *has to* be `0`, the source and target scopes `sx` and `sy` *have to* be the empty snoclists, and the proof `prf` of the invariant has to be of a specific shape. Rewriting by these equalities changes the goal type enough for the typechecker to ultimately see that the thinning was constructed using the `done` smart constructor and so we can use the view's `Done` constructor.

```
view (MkTh 0 bs prf) =
  let 0 eqs = isDone prf in
  rewrite bsIsZero eqs in
  rewrite fstIndexIsLin eqs in
  rewrite sndIndexIsLin eqs in
  rewrite invariantIsDone eqs in
  Done
```

In case the thinning is non-empty, we need to inspect the 0-th bit of the encoding to know whether it keeps or discards its most local variable. This is done by calling the `choose` function which takes a boolean `b` and returns a value of type (`Either` (`So` `b`) (`So` (`not` `b`))) i.e. we not only inspect the boolean but also record which value we got in a proof using the `So` family introduced in section 3.

```
view (MkTh (S i) bs prf) = case choose (testBit bs Z) of
```

If the bit is set then we know the variable is kept. And so we can invoke an inversion lemma that will once again provide us with a lot of equalities that we immediately deploy to reshape the goal's type. This ultimately lets us assemble a sub-thinning and use the view's `Keep` constructor.

```
  Left so =>
    let 0 eqs = isKeep prf so in
    rewrite fstIndexIsSnoc eqs in
    rewrite sndIndexIsSnoc eqs in
    rewrite invariantIsKeep eqs in
    rewrite isKeepInteger bs so in
    let th : Th eqs.fstIndexTail eqs.sndIndexTail
        th = MkTh i (bs `shiftR` 1) eqs.subInvariant in
    cast $ Keep th eqs.keptHead
```

If the bit is not set then we learn that the thinning was constructed using `drop`. We can once again use an inversion lemma to rearrange the goal and finally invoke the view's `Drop` constructor.

```
  Right soNot =>
    let 0 eqs = isDrop prf soNot in
    rewrite sndIndexIsSnoc eqs in
    rewrite invariantIsDrop eqs in
    rewrite isDropInteger bs soNot in
    let th : Th sx eqs.sndIndexTail
        th = MkTh i (bs `shiftR` 1) eqs.subInvariant in
    cast $ Drop th eqs.keptHead
```

We can readily use this function to implement pattern matching functions taking a thinning apart. We can for instance define `kept`, the function that counts the number of `keep` smart constructors used when manufacturing the input thinning and returns a proof that this is exactly the length of the source scope sx.

```
kept : Th sx sy -> (n : Nat ** length sx === n)
kept th = case view th of
  Done      => (0 ** Refl)
  Keep th x => let (n ** eq) = kept th in
               (S n ** cong S eq)
  Drop th x => kept th
```

We proceed by calling the view function on the input thinning which immediately tells us that we only have three cases to consider. The Done case is easily handled because the branch's refined types inform us that both sx and sy are the empty snoclist [<] whose length is evidently 0. In the Keep branch we learn that sx has the shape (_ :< x) and so we must return the successor of whatever the result of the recursive call gives us. Finally in the Drop case, sx is untouched and so a simple recursive call suffices. Note that the function is correctly detected as total because the target scope sy is indeed getting structurally smaller at every single recursive call. It is runtime irrelevant but it can still be successfully used as a termination measure by the compiler.

5.3 The Invariant Relation

We have shown the user-facing Th and have claimed that it is possible to define smart constructors done, keep, and drop, as well as a view function. This should become apparent once we show the actual definition of Invariant.

Definition of Invariant The relation maintains the invariant between the record's fields bigEnd (a Nat) and encoding (an Integer) and the index scopes sx and sy. Its definition can favour ease-of-use of runtime efficiency because we statically know that all of the Invariant proofs will be erased during compilation.

```
data Invariant : (i : Nat) -> (bs : Integer) ->
                 (sx, sy : SnocList a) -> Type where
  Done : Invariant Z 0 [<] [<]
  Keep : Invariant i (bs `shiftR` 1) sx sy -> (0 x : a) ->
         {auto 0 b  : So (testBit bs Z)} ->
         Invariant (S i) bs (sx :< x) (sy :< x)
  Drop : Invariant i (bs `shiftR` 1) sx sy -> (0 x : a) ->
         {auto 0 nb : So (not (testBit bs Z))} ->
         Invariant (S i) bs sx (sy :< x)
```

As always, the Done constructor is the simplest. It states that the thinning of size Z and encoded as the bit pattern 0 is the empty thinning.

The Keep constructor guarantees that the thinning of size (S i) and encoding bs represents an injection from (sx :< x) to (sy :< x) provided that the bit at position `Z` of `bs` is set, and that the rest of the bit pattern (obtained by a right shift on `bs`) is a valid thinning of size `i` from `sx` to `sy`.

The `Drop` constructor is structured the same way, except that it insists the bit at position `Z` should *not* be set.

We can readily use this relation to prove that some basic encoding are valid representations of useful thinnings.

Examples of `Invariant` proofs For instance, we can always define a thinning from the empty scope to an arbitrary scope `sy`.

```
  none : (sy : SnocList a) -> Th [<] sy
  none sy = MkTh (length sy) 0 (none sy)
```

The `encoding` of this thinning is `0` because every variable is being discarded and its `bigEnd` is the length of the outer scope `sy`. The validity proof is provided by the `none` lemma proven below. We once again use Idris 2's overloading to give the same to functions that play similar roles but at different types.

```
none : (sy : SnocList a) -> Invariant (length sy) 0 [<] sy
none [<] = Done
none (sy :< y) = Drop (none sy) y
```

The proof proceeds by induction over the outer scope `sy`. If it is empty, we can simply use the constructor for the empty thinning. Otherwise we can invoke `Drop` on the induction hypothesis. This all typechecks because (`testBit 0 Z`) computes to `False` and so the `nb` proof can be constructed automatically by Idris 2's proof search (cf. section 3.2), and (`0 ‘shiftR‘ 1`) evaluates to `0` which means the induction hypothesis has exactly the right type.

The definition of the identity thinning is a bit more involved. For a scope of size n, we are going to need to generate a bit pattern consisting of n ones. We define it in two steps. First, `cofull` defines a bit pattern of k zeros followed by infinitely many ones by shifting k places to the left a bit pattern of ones only. Then, we obtain `full` by taking the complement of `cofull`.

```
cofull : Nat -> Integer                    full : Nat -> Integer
cofull n = oneBits ‘shiftL‘ n              full n = complement (cofull n)
```

We can then define the identity thinning for a scope of size n by pairing (`full n`) as the `encoding` and `n` as the `bigEnd`.

```
  ones : (sx : SnocList a) -> Th sx sx
  ones sx = let n : Nat; n = length sx in MkTh n (full n) (ones sx)
```

The bulk of the work is once again in the eponymous lemma proving that this encoding is valid.

```
ones : (sx : SnocList a) ->
       let n = length sx in Invariant n (full n) sx sx
ones [<] = Done
ones (sx :< x) =
  let 0 nb = eqToSo (testBitFull (S (length sx)) Z) in
  Keep (rewrite shiftRFull (length sx) in ones sx) x
```

This proof proceeds once more by induction on the scope. If the scope is empty then once again the constructor for the empty thinning will do. In the non-empty case, we first appeal to an auxiliary lemma (not shown here) to construct a proof `nb` that the bit at position `Z` for a non-zero `full` integer is known to be `True`. We then need to use another lemma to cast the induction hypothesis which mentions (`full` (`length sx`)) so that it may be used in a position where we expect a proof talking about (`full` (`length` (`sx :< x`)) `` `shiftR` `` `1`).

Properties of the `Invariant` relation This relation has a lot of convenient properties.

First, it is proof irrelevant: any two proofs that the same `i`, `bs`, `sx`, and `sy` are related are provably equal. Consequently, equality on `Th` values amounts to equality of the `bigEnd` and `encoding` values. In particular it is cheap to test whether a given thinning is the empty or the identity thinning.

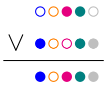
Second, it can be inverted [12] knowing only two bits: whether the natural number is empty and what the value of the bit at position `Z` of the encoding is. This is what allowed us to efficiently implement the `view` function by using these two checks and then inverting the `Invariant` proof to gain access to the proof that the remainder of the thinning's encoding is valid. We will see in section 5.5 that this leads to efficient runtime code for the view.

5.4 Choose Your Own Abstraction Level

Access to both the high-level `View` and the internal `Invariant` relation means that programmers can pick the level of abstraction at which they want to work. They may need to explicitly manipulate bits to implement key operators that are used in performance-critical paths but can also stay at the highest level for more negligible operations, or when proving runtime irrelevant properties.

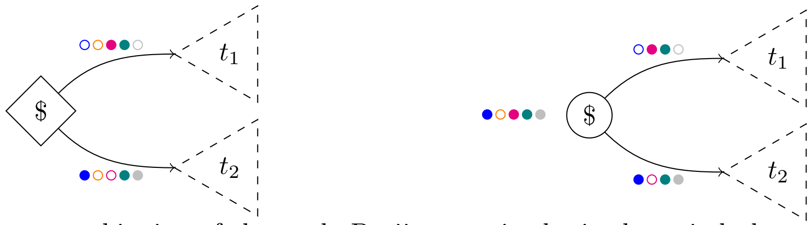
In the previous section we saw simple examples of these bit manipulations when defining `none` (using the constant 0 bit pattern) and `ones` using bit shifting and complement to form an initial segment of 1s followed by 0s.

Other natural examples include the *meet* and *join* of two thinnings sharing the same wider scope. The join can for instance be thought of either as a function defined by induction on the first thinning and case analysis on the second, emitting a `Keep` constructor whenever either of the inputs does. Or we can observe that the bit pattern in the join is the disjunction of the inputs' bit patterns and prove a lemma about the `Invariant` relation instead. This can be visualised as follows: in each column the meet is a • whenever either of the inputs is.



The join is of particular importance because it appears when we convert an ‘opened’ view of a term into its co-de Bruijn counterpart. As we mentioned earlier, co-de Bruijn terms in an arbitrary scope are represented by the pairing of a term indexed by its precise support with a thinning embedding this support back into the wider scope. When working with such a representation, it is convenient to have access to an ‘opened’ view where the outer thinning has been pushed inside therefore exposing the term’s top-level constructor, ready for case-analysis.

The following diagram shows the correspondence between an ‘opened’ application node using the view (the diamond ‘$’ node) with two subterms both living in the outer scope and its co-de Bruijn form (the circular ‘$’ node) with an outer thinning selecting the term support.



The outer thinning of the co-de Bruijn term is obtained precisely by computing the join of the respective outer thinnings of the ‘opened’ application’s function and argument.

These explicit bit manipulations will be preserved during compilation and thus deliver more efficient code.

5.5 Compiled Code

The following code block shows the JavaScript code that is produced when compiling the `view` function. We chose to use the JavaScript backend rather than e.g. the ChezScheme one because it produces fairly readable code. We have modified the backend to also write comments reminding the reader of the type of the function being defined and the data constructors the natural number tags correspond to. These changes are now available to all in Idris 2 version 0.6.0.

The only manual modifications we have performed are the inlining of a function corresponding to a **`case`** block, renaming variables and property names to make them human-readable, introducing the `$tail` definitions to make lines shorter, and slightly changing the layout.

```
/* Thin.Smart.view : (th : Th sx sy) -> View th */
function Thin_Smart_view($th) {
  switch($th.bigEnd) {
    case 0n: return {h: 0 /* Done */};
    default: {
      const $predBE = ($th.bigEnd-1n);
      const $test = choose(notEq(($th.encoding&1n), 0n)));
      switch($test.tag) {
        case 0: /* Left */ {
          const $tail = $th.encoding>>1n;
          return { tag: 1 /* Keep */
                 , val: {bigEnd: $predBE, encoding: $tail}}; }
        case 1: /* Right */ {
          const $tail = $th.encoding>>1n;
          return { tag: 2 /* Drop */
                 , val: {bigEnd: $predBE, encoding: $tail}}; }
}}}}
```

Readers can see that the compilation process has erased all of the indices and the proofs showing that the invariant tying the efficient runtime representation to the high-level specification is maintained. A thinning is represented at runtime by a JavaScript object with two properties corresponding to `Th`'s runtime relevant fields: `bigEnd` and `encoding`. Both are storing a JavaScript `bigInt` (one corresponding to the `Nat`, the other to the `Integer`). For instance the thinning [01101] would be at runtime `{ bigEnd: 5n, encoding: 13n }`.

The view proceeds in two steps. First if the `bigEnd` is `0n` then we know the thinning is empty and can immediately return the `Done` constructor. Otherwise we know the thinning to be non-empty and so we can compute the big end of its tail (`$predBE`) by subtracting one to the non-zero `bigEnd`. We can then inspect the bit at position `0` to decide whether to return a `Keep` or a `Drop` constructor. This is performed by using a bit mask to 0-out all the other bits (`$th.bigEnd&1n`) and checking whether the result is zero. If it is not equal to 0 then we emit `Keep` and compute the `$tail` of the thinning by shifting the original encoding to drop the 0th bit. Otherwise we emit `Drop` and compute the same tail.

By running `view` on this [01101] thinning, we would get back (`Keep` [0110]), that is to say `{ tag: 1, val: { bigEnd: 4n, encoding: 6n } }`.

Thanks to Idris 2's implementation of Quantitative Type Theory we have managed to manufacture a high level representation that can be manipulated like a classic inductive family using smart constructors and views without giving up an inch of control on its runtime representation.

The remaining issues such as the fact that we form the view's constructors only to immediately take them apart thus creating needless allocations can be tackled by reusing Wadler's analysis (section 12 of [24]).

6 Conclusion

We have seen that inductive families provide programmers with ways to root out bugs by enforcing strong invariants. Unfortunately these families can get in the way of producing performant code despite existing optimisation passes erasing redundant or runtime irrelevant data. This tension has led us to take advantage of Quantitative Type Theory in order to design a library combining the best of both worlds: the strong invariants and ease of use of inductive families together with the runtime performance of explicit bit manipulations.

6.1 Related Work

For historical and ergonomic reasons, idiomatic code in Coq tends to center programs written in a subset of the language quite close to OCaml and then prove properties about these programs in the runtime irrelevant `Prop` fragment. This can lead to awkward encodings when the unrefined inputs force the user to consider cases which ought to be impossible. Common coping strategies involve relaxing the types to insert a modicum of partiality e.g. returning an option type or taking an additional input to be used as the default return value. This approach completely misses the point of type-driven development. We benefit from having as much information as possible available during interactive editing. This information not only helps tremendously getting the definitions right by ensuring we always maintain vital invariants thus making invalid states unrepresentable, it also gives programmers access to type-driven tools and automation. Thankfully libraries such as Equations [20,21] can help users write more dependently typed programs, by taking care of the complex encoding required in Coq. A view-based approach similar to ours but using `Prop` instead of the zero quantity ought to be possible. We expect that the views encoded this way in Coq will have an even worse computational behaviour given that Equations uses a sophisticated elaboration process to encode dependent pattern-matching into Gallina. However Coq does benefit from good automation support for unfolding lemmas, inversion principles, and rewriting by equalities. It may compensate for the awkwardness introduced by the encoding.

Prior work on erasure [22] has the advantage of offering a fully automated analysis of the code. The main inconvenience is that users cannot state explicitly that a piece of data ought to be runtime irrelevant and so they may end up inadvertently using it which would prevent its erasure. Quantitative Type Theory allows us users to explicitly choose what is and is not runtime relevant, with the quantity checker keeping us true to our word. This should ensure that the resulting program has a much more predictable complexity.

A somewhat related idea was explored by Brady, McKinna, and Hammond in the context of circuit design [7]. In their verification work they index an efficient representation (natural numbers as a list of bits) by its meaning as a unary natural number. All the operations are correct by construction as witnessed by the use of their unary counterparts acting as type-level specifications. In the end their algorithms still process the inductive family instead of working directly with binary numbers. This makes sense in their setting where they construct circuits and so are explicitly manipulating wires carrying bits. By contrast, in our motivating example we really want to get down to actual (unbounded) integers rather than linked lists of bits.

6.2 Limitations and Future Work

Overall we found this case study using Idris 2, a state of the art language based on Quantitative Type Theory, very encouraging. The language implementation is still experimental but none of the issues are intrinsic limitations. We hope to be able to push this line of work further, tackling the following limitations and exploring more advanced use cases.

Limitations Unfortunately it is only *propositionally* true that (`view` (`keep th x`)) computes to (`Keep th x`) (and similarly for `done`/`Done` and `drop`/`Drop`). This means that users may need to manually deploy these lemmas when proving the properties of functions defined by pattern matching on the result of calling the `view` function. This annoyance would disappear if we had the ability to extend Idris 2's reduction rules with user-proven equations as implemented in Agda and formally studied by Cockx, Tabareau, and Winterhalter [10].

In this paper's case study, we were able to design the core `Invariant` relation making the invariants explicit in such a way that it would be provably proof irrelevant. This may not always be possible given the type theory currently implemented by Idris 2. Adding support for a proof-irrelevant sort of propositions (see e.g. Altenkirch, McBride, and Swierstra's work [3]) could solve this issue once and for all.

The Idris 2 standard library thankfully gave us access to a polished pure interface to explicitly manipulate an integer's bits. However these built-in operations came with no built-in properties whatsoever. And so we had to postulate a (minimal) set of axioms and prove a lot of useful corollaries ourselves. There is even less support for other low-level operations such as reading from a read-only array, or manipulating pointers.

We also found the use of runtime irrelevance (the **`0`** quantity) sometimes frustrating. Pattern-matching on a runtime irrelevant value in a runtime relevant context is currently only possible if it is manifest for the compiler that the value could only arise using one of the family's constructors. In non-trivial cases this is unfortunately only merely provable rather than self-evident. Consequently we are forced to jump through hoops to appease the quantity checker, and end up defining complex inversion lemmas to bypass these limitations. This could be solved by a mix of improvements to the typechecker and meta-programming using prior ideas on automating inversion [12,15,19].

Future work We are planning to explore more memory-mapped representations equipped with a high level interface.

We already have experimental results demonstrating that we can use a read-only array as a runtime representation of a binary search tree. Search can be implemented as a proven-correct high level decision procedure that is seemingly recursively exploring the "tree". At runtime however, this will effectively execute like a classic search by dichotomy over the array.

More generally, we expect that a lot of the work on programming on serialised data done in LoCal [23] thanks to specific support from the compiler can be done as-is in a QTT-based programming language. Indeed, QTT's type system is powerful enough that tracking these invariants can be done purely in library code.

In the short term, we would like to design a small embedded domain specific language giving users the ability to more easily build and take apart products and sums efficiently represented in the style we presented here. Staging would help here to ensure that the use of the eDSL comes at no runtime cost. There are plans to add type-enforced staging to Idris 2, thus really making it the ideal host language for our project.

Our long term plan is to go beyond read-only data and look at imperative programs proven correct using separation logic and see how much of this after-the-facts reasoning can be brought back into the types to enable a high-level correct-by-construction programming style that behaves the same at runtime.

Acknowledgements We are grateful to Conor McBride for discussions pertaining to the fine details of the unsafe encoding used in TypOS, as well as James McKinna, Fredrik Nordvall Forsberg, Ohad Kammar, and Jacques Carette for providing helpful comments and suggestions on early versions of this paper.

This research was funded by the Engineering and Physical Sciences Research Council (grant number EP/T007265/1).

The research data underpinning this publication [1] can be accessed at https://doi.org/10.17630/bd1085ce-a462-4a8b-ae81-9ededb4aea21.

References

1. Allais, G.: Builtin Types viewed as Inductive Families (code). University of St Andrews Research Portal (2023). https://doi.org/10.17630/bd1085ce-a462-4a8b-ae81-9ededb4aea21
2. Allais, G., Altenmüller, M., McBride, C., Nakov, G., Forsberg, F.N., Roy, C.: TypOS: An operating system for typechecking actors. In: 28th International Conference on Types for Proofs and Programs, TYPES 2022, June 20-25, 2022, Nantes, France (2022)
3. Altenkirch, T., McBride, C., Swierstra, W.: Observational equality, now! In: Stump, A., Xi, H. (eds.) Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007. pp. 57–68. ACM (2007). https://doi.org/10.1145/1292597.1292608
4. Atkey, R.: Syntax and semantics of quantitative type theory. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 56–65. ACM (2018). https://doi.org/10.1145/3209108.3209189
5. Brady, E.C.: Idris 2: Quantitative type theory in practice. In: Møller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference). LIPIcs, vol. 194, pp. 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.9
6. Brady, E.C., McBride, C., McKinna, J.: Inductive families need not store their indices. In: Berardi, S., Coppo, M., Damiani, F. (eds.) Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4,

2003, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3085, pp. 115–129. Springer (2003). https://doi.org/10.1007/978-3-540-24849-1_8

7. Brady, E.C., McKinna, J., Hammond, K.: Constructing correct circuits: Verification of functional aspects of hardware specifications with dependent types. In: Morazán, M.T. (ed.) Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4. 2007. Trends in Functional Programming, vol. 8, pp. 159–176. Intellect (2007)
8. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae (Proceedings) **75**(5), 381–392 (1972). https://doi.org/10.1016/1385-7258(72)90034-0, https://www.sciencedirect.com/science/article/pii/1385725872900340
9. Chapman, J.M.: Type checking and normalisation. Ph.D. thesis, University of Nottingham (July 2009), http://eprints.nottingham.ac.uk/10824/
10. Cockx, J., Tabareau, N., Winterhalter, T.: The taming of the rew: a type theory with computational assumptions. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434341
11. Coq Development Team, T.: The Coq Proof Assistant Reference Manual, version 8.15.2 (May 2022), http://coq.inria.fr
12. Cornes, C., Terrasse, D.: Automating inversion of inductive predicates in coq. In: Berardi, S., Coppo, M. (eds.) Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers. Lecture Notes in Computer Science, vol. 1158, pp. 85–104. Springer (1995). https://doi.org/10.1007/3-540-61780-9_64
13. Dybjer, P.: Inductive families. Formal Aspects Comput. **6**(4), 440–465 (1994). https://doi.org/10.1007/BF01211308
14. Maziarz, K., Ellis, T., Lawrence, A., Fitzgibbon, A.W., Jones, S.P.: Hashing modulo alpha-equivalence. 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. 960–973. ACM (2021). https://doi.org/10.1145/3453483.3454088
15. McBride, C.: Inverting inductively defined relations in LEGO. In: Giménez, E., Paulin-Mohring, C. (eds.) Types for Proofs and Programs, International Workshop TYPES'96, Aussois, France, December 15-19, 1996, Selected Papers. Lecture Notes in Computer Science, vol. 1512, pp. 236–253. Springer (1996). https://doi.org/10.1007/BFb0097795
16. McBride, C.: I got plenty o' nuttin'. In: Lindley, S., McBride, C., Trinder, P.W., Sannella, D. (eds.) A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 9600, pp. 207–233. Springer (2016). https://doi.org/10.1007/978-3-319-30936-1_12
17. McBride, C.: Everybody's got to be somewhere. In: Atkey, R., Lindley, S. (eds.) Proceedings of the 7th Workshop on Mathematically Structured Functional Programming, MSFP@FSCD 2018, Oxford, UK, 8th July 2018. EPTCS, vol. 275, pp. 53–69 (2018). https://doi.org/10.4204/EPTCS.275.6
18. McBride, C., McKinna, J.: The view from the left. J. Funct. Program. **14**(1), 69–111 (2004). https://doi.org/10.1017/S0956796803004829
19. Monin, J.F.: Proof Trick: Small Inversions. In: Bertot, Y. (ed.) Second Coq Workshop. Yves Bertot, Edinburgh, United Kingdom (Jul 2010), https://hal.inria.fr/inria-00489412

20. Sozeau, M.: Equations: A dependent pattern-matching compiler. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6172, pp. 419–434. Springer (2010). https://doi.org/10.1007/978-3-642-14052-5_29
21. Sozeau, M., Mangin, C.: Equations reloaded: high-level dependently-typed functional programming and proving in coq. Proc. ACM Program. Lang. **3**(ICFP), 86:1–86:29 (2019). https://doi.org/10.1145/3341690
22. Tejiščák, M.: A dependently typed calculus with pattern matching and erasure inference. Proc. ACM Program. Lang. **4**(ICFP), 91:1–91:29 (2020). https://doi.org/10.1145/3408973
23. Vollmer, M., Koparkar, C., Rainey, M., Sakka, L., Kulkarni, M., Newton, R.R.: Local: a language for programs operating on serialized data. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 48–62. ACM (2019). https://doi.org/10.1145/3314221.3314631
24. Wadler, P.: Views: A way for pattern matching to cohabit with data abstraction. In: Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987. pp. 307–313. ACM Press (1987). https://doi.org/10.1145/41625.41653
25. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989. pp. 60–76. ACM Press (1989). https://doi.org/10.1145/75277.75283

Pragmatic Gradual Polymorphism with References

Wenjia Ye[1] (✉) and Bruno C. d. S. Oliveira[1]

The University of Hong Kong, Hong Kong SAR, China
{wjye,bruno}@cs.hku.hk

Abstract. Gradualizing System F has been widely discussed. A big challenge is to preserve relational parametricity and/or the gradual guarantee. Most past work has focused on the preservation of parametricity, but often without the gradual guarantee. A few recent works satisfy both properties by giving up System F syntax, or with some restrictions and the introduction of sophisticated mechanisms in the dynamic semantics.

While parametricity is important for polymorphic languages, most mainstream languages typically do not satisfy it, for a variety of different reasons. In this paper, we explore the design space of polymorphic languages that satisfy the gradual guarantee, but do not preserve parametricity. When parametricity is not a goal, the design of polymorphic gradual languages can be considerably simplified. Moreover, it becomes easy to add features that are of practical importance, such as mutable references. We present a new gradually typed polymorphic calculus, called λ^G_{gpr}, with mutable references and with an easy proof of the gradual guarantee. In addition, compared to other gradual polymorphism work, λ^G_{gpr} is defined using a Type-Directed Operational Semantics (TDOS), which allows the dynamic semantics to be defined directly instead of elaborating to a target cast language. λ^G_{gpr} and all the proofs in this paper are formalized in Coq.

Keywords: Gradual Typing · Type System · Polymorphism.

1 Introduction

Statically typed languages can statically detect potential errors in programs, but must necessarily be conservative and reject some well-behaved programs. With dynamically typed languages, all programs are accepted, which offers a great amount of flexibility. However, the accepted dynamic programs include programs with type errors, making it harder to detect programs that are ill-behaved because of type errors. Considering the weaknesses and advantages of static and dynamic type systems, many approaches have proposed to integrate these two spectrums [1,7,35,22,8]. *Gradual typing* [31,35] provides a smooth integration of the two styles and has been under active research in the programming languages community. In addition to the type soundness property, a gradual language should behave as a static language if it is fully annotated. Conversely, it should behave as a dynamic language for fully dynamic programs. Importantly, the *gradual guarantee* [32] has been proposed to ensure a smooth transition between static and dynamic typing.

The importance of System F as a foundation for programming languages with polymorphism naturally leads to the question of whether it is possible to gradualize it. Various researchers have explored this question. In this line of research, a long-standing goal has been how to preserve *relational parametricity* [28]. Parametricity ensures a uniform behavior for all instantiations of polymorphic functions, and is an important property of System F. In addition it is also desirable to preserve the *gradual guarantee* [32], which is recognized as an important property for gradual languages. Unlike System F, where no dynamic mechanism is needed to ensure parametricity, with gradualized versions of System F this is no longer the case. Ahmed *et al.* [3] showed that parametricity can be enforced using a *dynamic sealing* mechanism at runtime. They prove parametricity, but the gradual guarantee is not discussed. Igarashi *et al.* [17] improved on the dynamic sealing approach and proposed a more efficient mechanism. While the gradual guarantee has been discussed, it was left as a conjecture. Toro *et al.* [37] even proved that gradual guarantee and parametricity are incompatible. By giving up the traditional System F syntax, New *et al.* [24] proved the gradual guarantee and parametricity by using user-provided sealing annotations, but this requires resorting to syntax that is not based on System F. Finally, Labrada *et al.* [20] proved the gradual guarantee and parametricity by inserting sealing with some restrictions. For instance, only base and variable types can be used to instantiate type applications.

While parametricity is highly valued and it is guaranteed in practice in some functional languages, many mainstream programming languages – such as Java, TypeScript or Flow – do not have parametricity. In mainstream languages the value of parametric polymorphism, and its ability to express a whole family of functions in a reusable and type-safe manner is certainly recognized. However, such languages are imperative and come with a variety of programming language features (such as unrestricted forms of mutable state, exceptions, parallelism and concurrency mechanisms, reflection, etc.) that make it hard to apply reasoning principles known in functional programming. In particular, most of those features are known to be highly challenging to deal with in the presence of parametricity [2,18,23]. This makes it non-obvious how to design a language with all those features, while preserving parametricity, in the first place. Moreover, preserving parametricity may require extra dynamic checks at runtime, which for implementations where performance is a critical factor may discourage implementers from doing such checks. Therefore all the aforementioned programming languages support System F like mechanisms to deal with polymorphism and benefit from the reuse afforded by polymorphism. However, the reasoning principles that arise from polymorphism, such as parametricity is discarded, and parametricity is not enforced.

In particular, programming languages such as TypeScript or Flow, which support some form of gradual/optional typing, and are widely used in practice, do not support parametricity. Figure 1 encodes an example from Ahmed *et al.*'s work [3], which was used to illustrate the parametricity challenge in gradual typing, in TypeScript and Flow. In this program, the polymorphic function `Ks` has a polymorphic type: $(X \to Y \to Y)$, where X and Y are type variables. In a calculus with parametricity, we know that a function with such type should always return the *second* argument or, in the presence of runtime casts, return an error. In the program, `Ks` is as a function that casts a dynamic constant function (`K`) that returns the *first* argument, which violates parametricity. When the TypeScript and Flow programs are run the first argument 2 is returned, illustrating that both languages do not enforce parametricity. In a gradual language with parametricity the result that we would expect is an error. Furthermore, even if we turn to Typed

```
function K(x:any, y:any): any {
    return x;
}

function Ks<X, Y>(x: X, y: Y): Y {
    let CAST = (K as any) as ((x:
  X, y: Y) ⇒ Y);
    return CAST(x, y);
}

function run() {
    console.log(Ks<number,
  number>(2,3));
}
```

(a) TypeScript code.

```
function K(x:any, y:any): any {
     return x;
}

function Ks<X, Y>(x: X, y: Y): Y {
    let CAST = ((K : any) : ((x:
  X, y: Y) ⇒ Y));
    return CAST(x, y);
}

function run() {
    console.log(Ks (2,3));

}
```

(b) Flow code.

Fig. 1: Ahmed *et al.* [3] program for illustrating parametricity in TypeScript and Flow.

Racket [36], which is a well-established gradual language used in both gradual typing research and in practice, the result is similar and 2 is returned:

```
(: K Any)
(define K ( λ (x) ( λ (y) x)))
(define Ks
  (cast K (All (X Y) (→ X (→ Y Y)))))
((Ks 2) 3)
```

Therefore Typed Racket does not enforce parametricity either.

In this paper, we explore the more pragmatic design space of polymorphic gradual languages with the gradual guarantee, but no parametricity. We believe that such designs are relevant because many practical language designs do not support parametricity, but support various other programming features instead. Dropping the requirement for parametricity enables us to explore language designs with many relevant practical features, while being in line with current designs for existing practical gradually typed languages. In particular, this paper studies the combination of parametric polymorphism, gradual typing and references. We show that, when parametricity is not a goal, the design of gradually polymorphic languages can be simplified, making it easier to add features such as references. Moreover, the gradual guarantee, which has shown to be quite problematic in all existing calculi with gradual polymorphism, is simple to achieve. We present a standard static calculus with polymorphism and mutable references called λ_{gpr}. Then we introduce the gradual counterpart, called λ^{G}_{gpr}.

The approach that we follow to give the dynamic semantics to λ^{G}_{gpr} is to use the recently proposed Type-Directed Operational Semantics TDOS [16,42]. In contrast, traditionally the semantics of a gradually typed language is defined by elaboration to a target *cast calculus* such as the *blame calculus* [39]. In other words, the dynamic semantics of the gradual source language is given *indirectly* by translating to the target language. As Ye *et al.* [42] shows, TDOS avoids such indirection and uses bidirectional typing and type annotations to enforce both *implicit* and *explicit* casting at runtime in gradually typed languages.

In summary, we make the following contributions in this paper:

- **The λ^G_{gpr} calculus:** A gradual calculus with polymorphism and mutable references. λ^G_{gpr} calculus is the gradual counterpart of the λ_{gpr} calculus. Both λ^G_{gpr} and λ_{gpr} are shown to be *type sound* and *deterministic*.
- **Gradual guarantee for λ^G_{gpr}.** We prove the *gradual guarantee* for λ^G_{gpr}. The proof is easy and quite simple, in contrast to previous work in gradual polymorphism, where the gradual guarantee was a major obstacle.
- **A TDOS extension.** TDOS has been applied to gradual typing before [42]. However, the previous work on TDOS for gradual typing only works in a purely functional, simply typed calculus. Our work shows that the TDOS approach can incorporate other features, including polymorphism and references.
- **A mechanical formalization in the Coq theorem prover.** All the calculi and proofs in this paper have been mechanically formalized in the Coq theorem prover. The Coq formalization can be found in the supplementary materials of this paper:

  `https://www.zenodo.org/badge/latestdoi/581421930`

2 Overview

This section provides a background for gradual polymorphic calculi, calculi with gradual references and the key ideas of our static system (λ_{gpr}) with polymorphism and references and its gradual counterpart (λ^G_{gpr}).

2.1 Background

Gradual References. Mutable references read or write content into a memory cell. A common set of operations is: allocating a memory cell (ref e); updating references ($e_1 := e_2$) and reading the content from a reference ($!e$). Locations (o) point to the memory cell. For a reference value ref 1, a new location (o) is generated and value 1 is stored in the cell at the location o. If 2 is assigned to this location $o := 2$, the cell value is updated to 2. Later, when we read this cell ($!o$), 2 is returned. Siek *et al.* [31] defined an *invariant* consistency relation for reference types. Reference types are only consistent with themselves. For example:

$$(\lambda x.\,(x := 2) : \text{Ref}\ \star \rightarrow \text{Ref}\ \star)\,(\text{ref}\ 1) \qquad -\ \text{Rejected!}\ \ \text{Ref}\ \mathsf{Int} \nsim \text{Ref}\ \star$$

Although the type Int is consistent with $\star$, it does not mean that Ref Int is consistent with Ref $\star$. Therefore, the argument type is not consistent with the function input, and the program is rejected. Herman *et al.* [14] proposed a gradually typed lambda source language with references, which defines the dynamic semantics by elaborating to a coercion calculus. The above program is allowed in their calculus. They define *variant* consistency where if A is consistent with B then Ref A is consistent with Ref B. In their calculus, casts are combined to achieve space-efficiency. Furthermore, Siek *et al.* [33] explored monotonic references with variant consistency. Their main consideration is space efficiency. No runtime overhead is imposed in the statically typed part of programs. All the above works have not considered the gradual guarantee.

Toro and Tanter [38] showed how to employ the Abstracting Gradual Typing (AGT) [12] methodology to design a gradually typed calculus with mutable references ($\lambda_{\widetilde{REF}}$). Their dynamic semantics of the source language is defined by translating to an evidence base calculus. They prove a bisimulation with the coercion calculus by Herman et al. [14]. $\lambda_{\widetilde{REF}}$ is proved to satisfy the gradual guarantee. The consistency of $\lambda_{\widetilde{REF}}$ is also variant.

Gradual Polymorphism. Gradual polymorphism is a popular topic. Researchers have been working in this area for a long time. Prior work has focused on two key properties: *relational parametricity* [28] and the *gradual guarantee* [32]. Relational parametricity ensures that all instantiations to a polymorphic value behave uniformly. The gradual guarantee ensures that less dynamic programs behave the same as more static programs.

Satisfying these two properties at once has shown to be problematic. Ahmed *et al.* [3] showed that a naive combination of the unknown type $\star$ and type substitution breaks relational parametricity. They show the problem using a simple expression with two casts. To simplify the presentation, we ignore blame labels. Suppose that $K^\star = \lceil \lambda x.\lambda y.x \rceil$, the dynamically typed constant function, is cast to a polymorphic type:

$$K^\star : \star \Rightarrow \forall X.\forall Y.X \to Y \to X \qquad\qquad K^\star : \star \Rightarrow \forall X.\forall Y.X \to Y \to Y$$

The notation $e : A \Rightarrow B$, borrowed from the blame calculus [29], means cast expression e from type A to type B. The constant function $K^\star$ returns the first argument. Considering relational parametricity, a value of type $\forall X.\forall Y.X \to Y \to X$ should be a constant value which always returns the first argument. While a value of type $\forall X.\forall Y.X \to Y \to Y$ should return the second argument. Therefore, the first cast succeeds and the second cast should fail. However, if these two casts are applied to the arguments in the usual way employing type substitutions, then we obtain the following:

$$\begin{aligned}
&(K^\star : \star \Rightarrow \forall X.\forall Y.X \to Y \to X)\,\mathsf{Int}\,\mathsf{Int}\,2\,3\\
\hookrightarrow^*\ &(K^\star : \star \Rightarrow \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int})\\
\hookrightarrow^*\ &2\\
&(K^\star : \star \Rightarrow \forall X.\forall Y.X \to Y \to Y)\,\mathsf{Int}\,\mathsf{Int}\,2\,3\\
\hookrightarrow^*\ &(K^\star : \star \Rightarrow \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int})\\
\hookrightarrow^*\ &2
\end{aligned}$$

The second cast succeeds and returns the first argument, which breaks parametricity. The reason for this behavior is that, after the type substitution, the polymorphic information is lost. Note that, as we have seen in Section 1, this is exactly how various practical languages (TypeScript, Flow and Typed Racket) behave.

Much of the work on gradual polymorphism aims at addressing the above problem. That is, for the second cast we would like to obtain blame instead of 2, so that parametricity is not violated. While the preservation of parametricity is a worthy goal, it typically requires substantial changes to a calculus to ensure its preservation, since naive direct type substitutions do not work. Furthermore, this also affects proofs, which can become significantly more complicated due to the changes in the calculus. To address this problem a well-known approach, originally proposed by Ahmed et al. [3], is to employ dynamic sealing. With dynamic sealing we do not do the substitution directly but record a fresh variable binding. However, even calculi that satisfy parametricity have to compromise on the important gradual guarantee property, or System F syntax, or be equiped with heavy forms of runtime evidence [37,20]. A thorough discussion of various approaches is given in Section 6.

2.2 Key Ideas

Our key design decision is to give up support for parametricity in exchange for a simpler calculus that is also easier to extend with other important practical features. In particular, in our work we illustrate how to obtain a polymorphic gradually typed calculus, with gradual references and with the gradual guarantee. In contrast, none of the existing gradually polymorphic calculi supports references and the gradual guarantee is only supported with restrictions [20]; or major modifications in the syntax and semantics of the language [24]; or not supported/proved at all [37,3,17].

A direct semantics with a TDOS. Our gradually typed calculus λ^G_{gpr} has a direct semantics by using a (TDOS) [15] approach. In λ^G_{gpr}, type annotations are *operationally relevant* and they basically play a role similar to casts. Nevertheless, implicit casts should also be enforced for a gradual calculus at runtime. Most previous work makes the implicit casts explicit via the elaboration process. That is the reason why dynamic semantics is not defined directly. We resort to bidirectional typing with inferred ($\Rightarrow$) and checked ($\Leftarrow$) modes. Using the checking mode of bidirectional typing, the consistency ($\sim$) between values and the checked type is checked and enforced via an implicit cast. At compile time, the flexible consistency relation allows more programs to be accepted, while the checking mode signals casts that are needed at runtime. For example, in the typing rule for applications.

$$\frac{\Sigma;\Gamma \vdash e_1 \;\Rightarrow\; A_1 \rightarrow A_2 \qquad \Sigma;\Gamma \vdash e_2 \;\Leftarrow\; A_1}{\Sigma;\Gamma \vdash e_1\, e_2 \;\Rightarrow\; A_2}\ \textsc{Typ-app}$$

The checking mode signals an implicit cast for the argument. The argument e_2 is checked to be consistent with the type A_1 using the bidirectional subsumption rule:

$$\frac{\Sigma;\Gamma \vdash e \;\Rightarrow\; B \qquad \Gamma \vdash B \sim A}{\Sigma;\Gamma \vdash e \;\Leftarrow\; A}\ \textsc{Typ-sim}$$

For instance, $(\lambda x.\, x : \mathsf{Int} \rightarrow \mathsf{Int})\,(\mathsf{True} : \star)$ type-checks, but at run-time the invalid cast to the value argument $(\mathsf{True} : \star)$ is detected and an error is reported.

Conservativity, no parametricity and direct substitutions. The λ^G_{gpr} calculus is a conservative extension of its static counterpart. Notably, our λ^G_{gpr} is a simple polymorphic calculus, without using mechanisms such as dynamic sealing and evidences. Instead, since parametricity is not a goal, we can simply use direct type substitutions during reduction as follows:

$$\frac{}{((\Lambda X.\, e : A) : \forall X.\, B)\ C \hookrightarrow e[X \mapsto C] : A[X \mapsto C] : B[X \mapsto C]}$$

Our type application rule substitutes type directly unlike in previous work with dynamic sealing where a fresh type name variable is generated and stored in a global or local context. Dynamic sealing takes extra time and space. With a large enough number of type applications, the space consumption may go unbounded.

Gradual guarantee and references. Furthermore, λ^{G}_{gpr} is mechanically formalized and shown to have the gradual guarantee. Our application of the eager semantics and the choice of value forms for λ^{G}_{gpr} simplify the gradual guarantee. To prove the gradual guarantee we need a precision ($\sqsubseteq$) relation. The gradual guarantee theorem needs to ensure that if the more static program does not go wrong, then the less static program should not go wrong as well. The precision relation is used to relate two programs, which have different type information. Type precision compares the amount of static type information for programs and types. A type is more precise than another if it is more static. The unknown type ($\star$) is the least precise type, since we do not have any static information about that type. Let's consider two programs:

$$\lambda x.\, 1 : \mathsf{Int} \to \mathsf{Int}$$
$$\lambda x.\, 1 : \star \to \star$$

The first one is more precise than the second one because the second program is fully dynamic. The value forms of λ^{G}_{gpr} are annotated and include terms such as $i : \mathsf{Int}$ and $(\lambda x.\, e : A \to B) : C$. The simplicity of the proof of the gradual guarantee is greatly related to the choice of representation of values. In λ^{G}_{gpr}, the gradual guarantee theorem can be formalized in a simple way with a lemma similar to a lemma proposed by Garcia et al. [12]. The lemma states that if e_1 is more precise than e_2 and e_1 takes a step to e'_1 then e_2 takes a step to e'_2 and e'_1 is more precise than e'_2. With this lemma, we can infer that two expressions related by precision have the same behavior. Thus, this lemma is enough to obtain the dynamic gradual guarantee. Notably, λ^{G}_{gpr} is extended with mutable references using a form of variant consistency [14,38]. This is in contrast to the previously discussed gradually polymorphic calculi where references are not supported.

3 The λ_{gpr} Calculus: Syntax, Typing and Semantics

In this section, we will introduce the λ_{gpr} calculus, which is a calculus with references and polymorphism. λ_{gpr} calculus is an extended version of System F with references and is the static calculus that serves as a foundation for the gradual calculus in Section 4.

3.1 Syntax

The syntax of the λ_{gpr} calculus is shown in Figure 2.

$\boxed{\text{Syntax}}$

Types	$A, B ::= \mathsf{Int} \mid A \to B \mid X \mid \forall X. A \mid \mathsf{Unit} \mid \mathsf{Ref}\, A$
Expressions	$e ::= x \mid i \mid \lambda x : A. e \mid e : A \mid e_1\, e_2 \mid \Lambda X. e \mid e\, A \mid !e \mid e_1 := e_2 \mid \mathsf{ref}\, e \mid \mathsf{unit} \mid o$
Values	$v ::= i \mid \Lambda X. e \mid \lambda x : A. e \mid \mathsf{unit} \mid o$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, X$
Stores	$\mu ::= \cdot \mid \mu, o = v$
Locations	$\Sigma ::= \cdot \mid \Sigma, o : A$
Frame	$F ::= v\, \square \mid \square\, e \mid \square\, A \mid !\, \square \mid v_1 := \square \mid \square := e_2 \mid \mathsf{ref}\, \square \mid \square : A$

Fig. 2: λ_{gpr} syntax

Types. Meta-variables A, B range over types. Types include base types (Int), function types ($A \to B$), type variables (X), polymorphic types ($\forall X. A$), the unit type Unit and reference types Ref A, which denotes a reference with type A.

Expressions. Meta-variables e range over expressions. Most of the expressions are standard: variables (x), integers (i), annotations ($e : A$), applications ($e_1\, e_2$), type applications ($e\, A$), dereferences ($!e$), assignments $e_1 := e_2$, references (ref e), unit (unit), locations o, lambda abstractions ($\lambda x : A. e$) (which are annotated with input type A), and type abstractions ($\Lambda X. e$).

Values. Meta-variables v range over values. A raw value is either an integer (i), a type abstraction ($\Lambda X. e$), a lambda abstraction ($\lambda x : A. e$), a unit (unit) or a location (o).

Contexts, stores, locations and frames. The type context Γ tracks the bound variables x with their types and the bound type variables X. Typing location Σ tracks the bound locations o with their types, while the store μ tracks locations with their stored values during the reduction process. Frames (F) include applications, type applications, dereferences, assignments and references.

3.2 Type System

Before introducing the type system, we show the well-formedness of types at the top of Figure 3. The well-formedness of types ensures that there are no free type variables and that each type variable is bound in the contexts.

Typing relation. The typing relation of λ_{gpr} is shown at the bottom of Figure 3. The type system essentially includes the usual System F rules, except that they also propagate the location typing context (Σ). Reference locations o are stored in the location typing context Σ (rule STYP-LOC). The bound type of locations indicates the type of stored values. For instance, o points to 1 stored in a memory cell. The integer type for 1 is tracked by the location o in the location typing context Σ. Other rules related to references such as assignments (rule STYP-ASSIGN), references (rule STYP-REF) and dereferences (rule STYP-DEREF) are standard. Annotation expressions ($e : A$) are not necessary for the static

$\boxed{\Gamma \vdash A}$ *(Well-formedness of types)*

$$\textsc{TW-int}\ \frac{}{\Gamma \vdash \mathsf{Int}} \qquad \textsc{TW-unit}\ \frac{}{\Gamma \vdash \mathsf{Unit}} \qquad \textsc{TW-var}\ \frac{X \in \Gamma}{\Gamma \vdash X} \qquad \textsc{TW-arr}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B} \qquad \textsc{TW-all}\ \frac{\Gamma, X \vdash A}{\Gamma \vdash \forall X. A} \qquad \textsc{TW-ref}\ \frac{\Gamma \vdash A}{\Gamma \vdash \mathsf{Ref}\, A}$$

$\boxed{\Sigma; \Gamma \vdash_s e : A}$ *(Typing rules for expressions)*

$$\textsc{STyp-lit}\ \frac{}{\Sigma; \Gamma \vdash_s i : \mathsf{Int}} \qquad \textsc{STyp-unit}\ \frac{}{\Sigma; \Gamma \vdash_s \mathsf{unit} : \mathsf{Unit}} \qquad \textsc{STyp-var}\ \frac{x : A \in \Gamma}{\Sigma; \Gamma \vdash_s x : A} \qquad \textsc{STyp-loc}\ \frac{o : A \in \Sigma}{\Sigma; \Gamma \vdash_s o : \mathsf{Ref}\, A}$$

$$\textsc{STyp-ref}\ \frac{\Sigma; \Gamma \vdash_s e : A}{\Sigma; \Gamma \vdash_s \mathsf{ref}\, e : \mathsf{Ref}\, A} \qquad \textsc{STyp-deref}\ \frac{\Sigma; \Gamma \vdash_s e : \mathsf{Ref}\, A}{\Sigma; \Gamma \vdash_s !e : A} \qquad \textsc{STyp-assign}\ \frac{\Sigma; \Gamma \vdash_s e_1 : \mathsf{Ref}\, A \qquad \Sigma; \Gamma \vdash_s e_2 : A}{\Sigma; \Gamma \vdash_s e_1 := e_2 : \mathsf{Unit}}$$

$$\textsc{STyp-abs}\ \frac{\Sigma; \Gamma, x : A \vdash_s e : B}{\Sigma; \Gamma \vdash_s \lambda x : A. e : A \to B} \qquad \textsc{STyp-app}\ \frac{\Sigma; \Gamma \vdash_s e_1 : A_1 \to A_2 \qquad \Sigma; \Gamma \vdash_s e_2 : A_1}{\Sigma; \Gamma \vdash_s e_1\, e_2 : A_2} \qquad \textsc{STyp-anno}\ \frac{\Sigma; \Gamma \vdash_s e : A}{\Sigma; \Gamma \vdash_s (e : A) : A}$$

$$\textsc{STyp-tabs}\ \frac{\Sigma; \Gamma, X \vdash_s e : A}{\Sigma; \Gamma \vdash_s \Lambda X. e : \forall X. A} \qquad \textsc{STyp-tapp}\ \frac{\Gamma \vdash A \qquad \Sigma; \Gamma \vdash_s e : \forall X. B}{\Sigma; \Gamma \vdash_s e\, A : B[X \mapsto A]}$$

Fig. 3: The type system of λ_{gpr} calculus.

system where the annotated types are syntactically equal (rule STYP-ANNO), but they will play an important role in the gradual system and are included here.
Definition 1 defines well-formed stores (μ) with respect to the typing locations Σ, using the typing relation:

Definition 1 (Well-formedness of the store with respect to Σ).

$$\Sigma \vdash \mu \equiv if\ dom(\mu) = dom(\Sigma)\ and\ \Sigma; \cdot \vdash \mu(o) : \Sigma(o),\ \ for\ every\ o \in \mu$$

A store is well-formed with the typing location if the store and the typing location contain the same domains. For each location, which is in the store, the bounded value $\mu(o)$ can be inferred with the type bound in the typing location ($\Sigma(o)$).

3.3 Dynamic Semantics

The operational semantics for the λ_{gpr} calculus is shown in Figure 4 (we ignore the gray parts for now). $\mu; e \hookrightarrow \mu'; e'$ represents the reduction rules, which states that e with store μ reduces to e' with the updated store μ'. The reduction rules of λ_{gpr} are

$\boxed{\mu; e \hookrightarrow_s \mu'; e'}$ *(Operational semantics)*

STEP-EVAL
$$\frac{\mu; e \hookrightarrow_s \mu'; e'}{\mu; F[e] \hookrightarrow_s \mu'; F[e']}$$

STEP-ANNOV
$$\frac{}{\mu; v : A \; : A \hookrightarrow_s \mu; v \; : A}$$

STEP-ASSIGN
$$\frac{}{\mu; o := v \hookrightarrow_s \mu[o \mapsto v]; \mathsf{unit}}$$

STEP-TAP
$$\frac{}{\mu; ((\Lambda X. e) \; : \forall X. A \;)\, A \hookrightarrow_s \mu; (e[X \mapsto A]) \; : (A[X \mapsto A])}$$

STEP-DEREF
$$\frac{o = v \in \mu}{\mu; !o \hookrightarrow_s \mu; v : A}$$

STEP-BETA
$$\frac{}{\mu; ((\lambda x : A. e) \; : A \to B \;)\, v \hookrightarrow_s \mu; e[x \mapsto v] \; : B \; : B}$$

STEP-REFV
$$\frac{o \notin \mu}{\mu; \mathsf{ref}\, v \hookrightarrow_s \mu, o = v; o}$$

Fig. 4: Reduction rules for λ_{gpr}.

straightforward. A reference value is bound in the store by a fresh location as shown in rule STEP-REFV. The dereference rule extracts the bound value of the location in the store (rule STEP-DEREF). Rule STEP-EVAL evaluates the frames. Let's see how the example $o_1 := (\Lambda X. (\lambda x : X. x)\, !o_2)\ \mathsf{Int}$ with the existing store $o_1 = 1, o_2 = 2$ reduces. 2 is read from store $o_1 = 1, o_2 = 2$. After the type substitution, 2 is substituted into the lambda. Then 2 is used to update the store pointed by o_1. Finally, the store becomes $o_1 = 2, o_2 = 2$. The detailed steps are as follows:

$$\begin{aligned}
& o_1 = 1, o_2 = 2; o_1 := (\Lambda X. (\lambda x : X. x)\, !o_2)\ \mathsf{Int} \\
\hookrightarrow\ & \{\text{by rule STEP-EVAL, rule STEP-DEREF}\} \\
& o_1 = 1, o_2 = 2; o_1 := (\Lambda X. (\lambda x : X. x)\, 2)\ \mathsf{Int} \\
\hookrightarrow\ & \{\text{by rule STEP-TAP}\} \\
& o_1 = 1, o_2 = 2; o_1 := (\lambda x : \mathsf{Int}. x)\, 2 \\
\hookrightarrow\ & \{\text{by rule STEP-BETA}\} \\
& o_1 = 1, o_2 = 2; o_1 := 2 \\
\hookrightarrow\ & \{\text{by rule STEP-ASSIGN}\} \\
& o_1 = 2, o_2 = 2; \mathsf{unit}
\end{aligned}$$

Theorem 1 shows that the λ_{gpr} calculus is deterministic:

Theorem 1 (Determinism of λ_{gpr}). *If $\Sigma; \cdot \vdash_s e : A$, $\Sigma \vdash \mu$, $\mu; e \hookrightarrow_s \mu_1; e_1$ and $\mu; e \hookrightarrow_s \mu_2; e_2$ then $e_1 = e_2$ and $\mu_1 = \mu_2$.*

Furthermore, the preservation Theorem 2 and progress Theorem 3 of λ_{gpr} calculus are shown below:

Theorem 2 (Type Preservation of λ_{gpr}). *If $\Sigma; \cdot \vdash_s e : A$, $\Sigma \vdash \mu$ and $\mu; e \hookrightarrow_s \mu'; e'$ then $\Sigma'; \cdot \vdash_s e' : A$, $\Sigma' \vdash \mu'$ and $\Sigma' \supseteq \Sigma$.*

Theorem 3 (Progress of λ_{gpr}). *If $\Sigma; \cdot \vdash_s e : A$ then e is a value or $\exists e' \mu'$, $\mu; e \hookrightarrow_s \mu'; e'$.*

$$\text{Typing modes} \qquad \Leftrightarrow ::= \ \Rightarrow | \Leftarrow$$

$\boxed{\Sigma; \Gamma \vDash_s e \Leftrightarrow A}$ *(Typing rules for expressions)*

$$\textsc{sty-lit}\ \frac{}{\Sigma; \Gamma \vDash_s i \Rightarrow \mathsf{Int}} \qquad \textsc{sty-unit}\ \frac{}{\Sigma; \Gamma \vDash_s \mathsf{unit} \Rightarrow \mathsf{Unit}} \qquad \textsc{sty-var}\ \frac{x : A \in \Gamma}{\Sigma; \Gamma \vDash_s x \Rightarrow A} \qquad \textsc{sty-loc}\ \frac{o : A \in \Sigma}{\Sigma; \Gamma \vDash_s o \Rightarrow \mathsf{Ref}\, A}$$

$$\textsc{sty-ref}\ \frac{\Sigma; \Gamma \vDash_s e \Rightarrow A}{\Sigma; \Gamma \vDash_s \mathsf{ref}\, e \Rightarrow \mathsf{Ref}\, A} \qquad \textsc{sty-deref}\ \frac{\Sigma; \Gamma \vDash_s e \Rightarrow \mathsf{Ref}\, A}{\Sigma; \Gamma \vDash_s !e \Rightarrow A} \qquad \textsc{sty-assign}\ \frac{\Sigma; \Gamma \vDash_s e_1 \Rightarrow \mathsf{Ref}\, A \quad \Sigma; \Gamma \vDash_s e_2 \Leftarrow A}{\Sigma; \Gamma \vDash_s e_1 := e_2 \Rightarrow \mathsf{Unit}}$$

$$\textsc{sty-abs}\ \frac{\Sigma; \Gamma, x : A \vDash_s e \Rightarrow B}{\Sigma; \Gamma \vDash_s \lambda x : A.\, e \Rightarrow A \to B} \qquad \textsc{sty-app}\ \frac{\Sigma; \Gamma \vDash_s e_1 \Rightarrow A_1 \to A_2 \quad \Sigma; \Gamma \vDash_s e_2 \Leftarrow A_1}{\Sigma; \Gamma \vDash_s e_1\, e_2 \Rightarrow A_2} \qquad \textsc{sty-anno}\ \frac{\Sigma; \Gamma \vDash_s e \Leftarrow A}{\Sigma; \Gamma \vDash_s e : A \Rightarrow A}$$

$$\textsc{sty-eq}\ \frac{\Sigma; \Gamma \vDash_s e \Rightarrow A}{\Sigma; \Gamma \vDash_s e \Leftarrow A} \qquad \textsc{sty-tabs}\ \frac{\Sigma; \Gamma, X \vDash_s e \Rightarrow A}{\Sigma; \Gamma \vDash_s \Lambda X.\, e \Rightarrow \forall X.\, A} \qquad \textsc{sty-tapp}\ \frac{\Gamma \vdash A \quad \Sigma; \Gamma \vDash_s e \Rightarrow \forall X.\, B}{\Sigma; \Gamma \vDash_s e\, A \Rightarrow B[X \mapsto A]}$$

Fig. 5: Bidirectional typing for the λ_{gpr} calculus.

3.4 Bidirectional Typing

We also present a set of bidirectional typing rules (shown in Figure 5) for λ_{gpr}. Although bidirectional typing is not essential for λ_{gpr}, it is used later for the gradual typing criteria proofs. The typing judgment is represented as $\Sigma; \Gamma \vdash e \Leftrightarrow A$. The expression e is inferred ($\Rightarrow$) or checked ($\Leftarrow$) by type A under the typing context Γ and location typing context Σ. Typing modes ($\Leftrightarrow$) contain the inference mode ($\Rightarrow$) and checking mode ($\Leftarrow$), which are shown at the top of Figure 5. One extra rule is rule STY-EQ, which switches modes. We proved that the two type systems are equivalent:

Lemma 1 (Typing Equivalence for λ_{gpr}). $\Sigma; \Gamma \vdash_s e : A$ *iff* $\Sigma; \Gamma \vDash_s e \Leftrightarrow A$.

4 The λ^G_{gpr} Calculus

This section introduces the λ^G_{gpr} calculus, which gradualizes the λ_{gpr} calculus. Normally, a gradually typed lambda calculus (GTLC) does not define the operational semantics directly, but is elaborated to a cast calculus. λ^G_{gpr} instead defines the dynamic semantics directly using the TDOS approach [15]. λ^G_{gpr} is proved to be type sound and it has a gradual guarantee. The calculus does not have parametricity, enabling simplifications

$\boxed{\text{Syntax}}$

Types	$A, B ::= \mathsf{Int} \mid A \to B \mid X \mid \forall X.\, A \mid \mathsf{Unit} \mid \mathsf{Ref}\, A \mid \star$
Expressions	$e ::= x \mid i \mid e : A \mid e_1\, e_2 \mid e\, A \mid !e \mid e_1 := e_2 \mid \mathsf{ref}\, e \mid \mathsf{unit} \mid o \mid \Lambda X.\, e : A \mid \lambda x.\, e : A \to B$
Results	$r ::= e \mid \mathsf{blame}$
Raw Values	$u ::= i \mid \Lambda X.\, e : A \mid \lambda x.\, e : A \to B \mid \mathsf{unit} \mid o$
Values	$v ::= u : A$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, X$
Stores	$\mu ::= \cdot \mid \mu, o = v$
Location	$\Sigma ::= \cdot \mid \Sigma, o : A$
Frame	$F ::= v\, \square \mid \square\, e \mid \square\, A \mid !\square \mid v_1 := \square \mid \square := e_2 \mid \mathsf{ref}\, \square$

$\boxed{\Gamma \vdash A \sim B}$ *(Consistency)*

$$\textsc{S-unit}\ \frac{}{\Gamma \vdash \mathsf{Unit} \sim \mathsf{Unit}} \qquad \textsc{S-var}\ \frac{\Gamma \vdash X}{\Gamma \vdash X \sim X} \qquad \textsc{S-z}\ \frac{}{\Gamma \vdash \mathsf{Int} \sim \mathsf{Int}} \qquad \textsc{S-dynL}\ \frac{\Gamma \vdash A}{\Gamma \vdash \star \sim A} \qquad \textsc{S-dynR}\ \frac{\Gamma \vdash A}{\Gamma \vdash A \sim \star}$$

$$\textsc{S-arr}\ \frac{\Gamma \vdash A_1 \sim B_1 \quad \Gamma \vdash A_2 \sim B_2}{\Gamma \vdash A_1 \to A_2 \sim B_1 \to B_2} \qquad \textsc{S-forall}\ \frac{\Gamma, X \vdash A \sim B}{\Gamma \vdash \forall X.\, A \sim \forall X.\, B} \qquad \textsc{S-ref}\ \frac{\Gamma \vdash A \sim B}{\Gamma \vdash \mathsf{Ref}\, A \sim \mathsf{Ref}\, B}$$

Fig. 6: λ^{G}_{gpr} syntax and consistency.

in the calculus, and the addition of features such as gradual references, which none of the previous gradual calculi with polymorphism support.

4.1 Static Semantics

Syntax, type well-formedness and consistency. Figure 6 shows the syntax and consistency of the λ^{G}_{gpr} calculus. The gray parts are the same as λ_{gpr}. The λ^{G}_{gpr} calculus extends types with the unknown type $\star$ with respect to λ_{gpr}. Because of the power of the unknown type $\star$, dynamic type checking is required and run-time errors may be raised. Therefore, in addition to expressions, λ^{G}_{gpr} has the run-time error blame. Because of the run-time checking requirement for the gradual typing system, we need annotations for type abstractions and lambda abstractions. Furthermore, due to the imprecision of the unknown type $\star$, values are also annotated. Otherwise, examples such as $1 : \star$ are troublesome. Because of the value forms, annotations are not included in frames, unlike in the λ_{gpr} calculus. We will explain the details later.

Well-formed types are extended with the following rule for the unknown type $\star$:

$$\frac{}{\Gamma \vdash \star}$$

Notably, instead of syntactic equality, a more general relation called consistency ($\Gamma \vdash A \sim B$) is defined in λ^{G}_{gpr}. Every well-formed type is consistent with itself. The unknown

$\boxed{\Sigma;\Gamma \vdash e \Leftrightarrow A}$ *(Typing rules for expressions)*

$$\textsc{Typ-lit}\quad \frac{}{\Sigma;\Gamma \vdash i \Rightarrow \mathsf{Int}} \qquad \textsc{Typ-unit}\quad \frac{}{\Sigma;\Gamma \vdash \mathsf{unit} \Rightarrow \mathsf{Unit}} \qquad \textsc{Typ-var}\quad \frac{x : A \in \Gamma}{\Sigma;\Gamma \vdash x \Rightarrow A} \qquad \textsc{Typ-loc}\quad \frac{o : A \in \Sigma}{\Sigma;\Gamma \vdash o \Rightarrow \mathsf{Ref}\,A}$$

$$\textsc{Typ-ref}\quad \frac{\Sigma;\Gamma \vdash e \Rightarrow A}{\Sigma;\Gamma \vdash \mathsf{ref}\,e \Rightarrow \mathsf{Ref}\,A} \qquad \textsc{Typ-deref}\quad \frac{A_1 \rhd \mathsf{Ref}\,A \quad \Sigma;\Gamma \vdash e \Rightarrow A_1}{\Sigma;\Gamma \vdash !e \Rightarrow A} \qquad \textsc{Typ-assign}\quad \frac{A_1 \rhd \mathsf{Ref}\,A \quad \Sigma;\Gamma \vdash e_1 \Rightarrow A_1 \quad \Sigma;\Gamma \vdash e_2 \Leftarrow A}{\Sigma;\Gamma \vdash e_1 := e_2 \Rightarrow \mathsf{Unit}}$$

$$\textsc{Typ-abs}\quad \frac{\Sigma;\Gamma, x : A \vdash e \Leftarrow B}{\Sigma;\Gamma \vdash \lambda x.\,e : A \to B \Rightarrow A \to B} \qquad \textsc{Typ-app}\quad \frac{A \rhd A_1 \to A_2 \quad \Sigma;\Gamma \vdash e_1 \Rightarrow A \quad \Sigma;\Gamma \vdash e_2 \Leftarrow A_1}{\Sigma;\Gamma \vdash e_1\,e_2 \Rightarrow A_2} \qquad \textsc{Typ-anno}\quad \frac{\Sigma;\Gamma \vdash e \Leftarrow A}{\Sigma;\Gamma \vdash e : A \Rightarrow A}$$

$$\textsc{Typ-sim}\quad \frac{\Gamma \vdash A \sim B \quad \Sigma;\Gamma \vdash e \Rightarrow A}{\Sigma;\Gamma \vdash e \Leftarrow B} \qquad \textsc{Typ-tabs}\quad \frac{\Sigma;\Gamma, X \vdash e \Leftarrow A}{\Sigma;\Gamma \vdash \Lambda X.\,e : A \Rightarrow \forall X.\,A} \qquad \textsc{Typ-tapp}\quad \frac{\Gamma \vdash A \quad \Sigma;\Gamma \vdash e \Rightarrow A_1 \quad A_1 \rhd \forall X.\,B}{\Sigma;\Gamma \vdash e\,A \Rightarrow B[X \mapsto A]}$$

$A \rhd A_1 \to A_2$	$A \rhd \forall X.\,A_1$	$A \rhd \mathsf{Ref}\,A_1$
$A \to B \rhd A \to B$	$\forall X.\,A \rhd \forall X.\,A$	$\mathsf{Ref}\,A \rhd \mathsf{Ref}\,A$
$\star \rhd \star \to \star$	$\star \rhd \forall X.\,\star$	$\star \rhd \mathsf{Ref}\,\star$

Fig. 7: The type system for the λ^G_{gpr} calculus.

type is consistent with any other well-formed type. Structural types such as functions, references and polymorphic types are consistent if their type sub-components are consistent. Note that for two reference types, consistency is variant: if A and B are consistent then Ref A and Ref B are consistent. Unlike invariant consistency [31], type A and B do not have to be the same. As usual, consistency is reflexive and symmetric, but not transitive. We use the following abbreviation for consistency: $A \sim B \equiv \cdot \vdash A \sim B$.

Typing relation. Bidirectional typing is used to design the type system. Bidirectional typing is not essential for λ_{gpr} but it is necessary for λ^G_{gpr}. Annotation expressions ($e : A$) and the checking mode ($\Leftarrow$) signal the use of casts (explicitly or implicitly) at run-time.

The typing rules of the λ^G_{gpr} calculus are shown in Figure 7. They are almost the same as λ_{gpr}'s type system. For rule Typ-app, rule Typ-tapp, rule Typ-assign and rule Typ-deref, the unknown type $\star$ can be matched with, respectively, a dynamic function type ($\star \to \star$), a dynamic polymorphic type ($\forall X.\,\star$) and a dynamic reference type (Ref $\star$). In a system with gradual typing and the unknown type $\star$ we always have to consider cases where the type may be unknown. For instance in an application $e_1\ e_2$, e_1 can infer a function type as usual, but it can also infer type $\star$ and still be well-typed. So, a matching function $(A \rhd B)$ is needed to account for both possibilities. The table at the bottom of Figure 7 shows the definition of the matching functions $A \rhd B$. Note that we overload the notation, but there are 3 different matching functions, in each column of the table, that are employed by the rules correspondingly. For example, rule Typ-deref employs the matching function in the third column of the table. The first row in the table depicts the form of the matching function, while the other two rows give its definition.

The checking mode rule Typ-sim is generalized to check if the inferred type A and checked type B are consistent. Note that rule Typ-sim is the only rule in the checked mode and, as such, does not overlap with anything else. Moreover, all the rules in the inference mode are syntax directed. Therefore, the rules are basically directly implementable, as usual for bidirectional type-checking rules. Note that in λ^G_{gpr} annotation expressions combined with consistency play an important role, where more programs are allowed. For instance, $(\lambda x.\,((x : \star)\,1) : \mathsf{Bool} \rightarrow \star)\,\mathsf{True}$ is accepted, but raises a blame error at run-time. Note that dynamically typed lambdas $\lambda x.e$ are syntactic sugar for $\lambda x.e : \star \rightarrow \star$. The use of this syntactic sugar enables us to encode the dynamically typed lambda calculus (DTLC) [4] easily in λ^G_{gpr}.

Definition 2 shows dynamic type checking for raw and annotated values, which is done at run-time. Dynamic type checking for values exploits the annotations that are present at run-time, and does not make use of the typing relation. Dynamic type checking is essentially a constant time operation, with little cost (note that the function is not recursive).

Definition 2 (Dynamic type). *$|u|_\mu = A$ and $|v|_\mu = A$ denote the dynamic type of the raw and annotated values.*

$$\begin{aligned}
|i|_\mu &= \mathsf{Int} \\
|(\lambda x.\,e : A \rightarrow B)|_\mu &= A \rightarrow B \\
|(\Lambda X.\,e : A)|_\mu &= \forall X.\,A \\
|\mathsf{unit}|_\mu &= \mathsf{Unit} \\
|o|_\mu &= \mathit{Ref}\,|v|_\mu \quad \textit{when } o = v \in \mu \\
|(u : A)|_\mu &= A
\end{aligned}$$

$|u|_\mu = A$ states that the dynamic type of the raw value u is A under store μ. Notably, for locations o, the dynamic type is defined by the dynamic type of the bounded values in the store. Other rules are straightforward. Lemma 2 shows that if a raw value can be inferred with type A, then its dynamic type is type A as well.

Lemma 2 (Synthesis of Dynamic Types). *For any raw value u, if $\Sigma \vdash \mu$ and $\Sigma; \cdot \vdash u \Rightarrow A$ then $|u|_\mu = A$.*

As in λ_{gpr}, a term typed using the inference mode is guaranteed to infer a unique type. In addition, Lemma 3 shows that each well-typed term can be checked.

Lemma 3 (Synthesis principality). *If $\Sigma; \Gamma \vdash e \Rightarrow A$ then exists B, $\Sigma; \Gamma \vdash e \Leftarrow B$ and $\Gamma \vdash A \sim B$.*

$\boxed{\mu; v \hookrightarrow_A \mu'; r}$ *(Casting for values)*

CASTING-SIM
$$\frac{|u|_\mu \sim B}{\mu; u : A \hookrightarrow_B \mu; u : B}$$

CASTING-NSIM
$$\frac{\neg|u|_\mu \sim B}{\mu; u : A \hookrightarrow_B \mu; \mathsf{blame}}$$

$\boxed{\mu; v \hookrightarrow_{B,A} \mu'; r}$ *(Double casting)*

TLISTS-BASEB
$$\frac{\mu; v \hookrightarrow_A \mu; \mathsf{blame}}{\mu; v \hookrightarrow_{B,A} \mu; \mathsf{blame}}$$

TLISTS-CONS
$$\frac{\mu; v \hookrightarrow_A \mu; v' \qquad \mu; v' \hookrightarrow_B \mu; r}{\mu; v \hookrightarrow_{B,A} \mu; r}$$

Fig. 8: Casting for values

4.2 Dynamic Semantics

The dynamic semantics contains two parts. The first part is casting, which casts a value to another value with a target type. In casting the dynamic type of the value is the source type. The second part is the reduction rules.

Casting. Figure 8 shows the casting rules of the λ^G_{gpr} calculus. $\mu; v \hookrightarrow_A \mu; r$ represents casting values v by type A under store μ. The dynamic type of the raw values u is checked to be consistent with type A or not. If two types are consistent, then the intermediate type can be removed and the raw values are annotated with target types. Otherwise, a run-time error is raised. For example when $1 : \star$ is cast by type Bool, the dynamic type of 1 is Int, which is not consistent with Bool, and blame is raised. While in $1 : \star$ cast by type Int, the type Int is consistent with type Int. Thus, type $\star$ is erased and 1 is annotated with type Int. Since a location o is a raw value, if we want to obtain the dynamic type of the location, we should obtain it from the store μ. Therefore, casting uses the store. Casting by two types is shown at the bottom of Figure 8. It simply casts the types one by one, using the basic casting relation.

Reduction. The reduction rules of λ^G_{gpr} calculus are shown in Figure 9. Raw values are reduced to become values, which are annotated by the dynamic type of the raw values with rule STEP-U. Due to this rule, annotations are not included in the frame. Annotated expressions are further dealt by rule STEP-ANNO and rule STEP-ANNOP. From the typing rules of rules TYP-APP, TYP-TAPP, TYP-ASSIGN, and TYP-DEREF, type $\star$ is allowed to match, respectively, a dynamic function, a polymorphic function or a reference type. Moreover, we know that $\star$ is consistent with any type. Therefore, we should check whether the internal values cannot match with the wanted type structure. For example, ill-formed applications $((1 : \star)\, 2)$ where the internal value (1) is not an lambda abstraction. There are similar examples for type applications and assignments: $(1 : \star)\ \mathsf{Bool}$ and $(\mathsf{True} : \star) := 2$ where 1 is not a type abstraction and True is not a location. Using

$\boxed{\mu; e \hookrightarrow \mu'; r}$ *(Operational semantics)*

$$\text{VSTEP-EVAL}\quad \frac{\mu; e \hookrightarrow \mu'; e'}{\mu; F[e] \hookrightarrow \mu'; F[e']}$$

$$\text{VSTEP-BLAME}\quad \frac{\mu; e \hookrightarrow \mu'; \mathsf{blame}}{\mu; F[e] \hookrightarrow \mu'; \mathsf{blame}}$$

$$\text{VSTEP-ANNOP}\quad \frac{\mu; e \hookrightarrow \mu'; \mathsf{blame}}{\mu; e : A \hookrightarrow \mu'; \mathsf{blame}}$$

$$\text{VSTEP-BETA}\quad \frac{A \rhd A_2 \to B_2 \qquad \mu; v \hookrightarrow_{A_1,A_2} \mu; v'}{\mu; ((\lambda x.\, e : A_1 \to B_1) : A)\, v \hookrightarrow \mu; e[x \mapsto v'] : B_1 : B_2}$$

$$\text{VSTEP-ASSIGND}\quad \frac{\mu; v_1 \hookrightarrow_{\mathrm{Ref}\,\star} \mu; \mathsf{blame}}{\mu; v_1 := v_2 \hookrightarrow \mu; \mathsf{blame}}$$

$$\text{VSTEP-ANNOV}\quad \frac{\mu; v \hookrightarrow_{A} \mu; r}{\mu; v : A \hookrightarrow \mu; r}$$

$$\text{VSTEP-BETAP}\quad \frac{A \rhd A_2 \to B_2 \qquad \mu; v \hookrightarrow_{A_1,A_2} \mu; \mathsf{blame}}{\mu; ((\lambda x.\, e : A_1 \to B_1) : A)\, v \hookrightarrow \mu; \mathsf{blame}}$$

$$\text{VSTEP-BETAD}\quad \frac{\mu; v_1 \hookrightarrow_{\star \to \star} \mu; \mathsf{blame}}{\mu; v_1\, v_2 \hookrightarrow \mu; \mathsf{blame}}$$

$$\text{VSTEP-TAP}\quad \frac{B \rhd \forall X.\, B_2}{\mu; ((\Lambda X.\, e : A) : B)\, C \hookrightarrow \mu; e[X \mapsto C] : A[X \mapsto C] : B_2[X \mapsto C]}$$

$$\text{VSTEP-TAPD}\quad \frac{\mu; v \hookrightarrow_{\forall X.\,\star} \mu; \mathsf{blame}}{\mu; v\, B \hookrightarrow \mu; \mathsf{blame}}$$

$$\text{VSTEP-REFV}\quad \frac{o \notin \mu}{\mu; \mathrm{ref}\, v \hookrightarrow \mu, o = v; o}$$

$$\text{VSTEP-DEREF}\quad \frac{o = v \in \mu \qquad A_1 \rhd \mathrm{Ref}\, A}{\mu; !(o : A_1) \hookrightarrow \mu; v : A}$$

$$\text{VSTEP-DEREFP}\quad \frac{\mu; v \hookrightarrow_{\mathrm{Ref}\,\star} \mu; \mathsf{blame}}{\mu; !v \hookrightarrow \mu; \mathsf{blame}}$$

$$\text{VSTEP-ASSIGN}\quad \frac{|o|_{\mu} = A_1 \qquad A \rhd \mathrm{Ref}\, A_2 \qquad \mu; v_2 \hookrightarrow_{A_1,A_2} \mu; v_2'}{\mu; (o : A) := v_2 \hookrightarrow \mu[o \mapsto v_2']; \mathsf{unit}}$$

$$\text{VSTEP-ASSIGNP}\quad \frac{|o|_{\mu} = A_1 \qquad A \rhd \mathrm{Ref}\, A_2 \qquad \mu; v_2 \hookrightarrow_{A_1,A_2} \mu; \mathsf{blame}}{\mu; (o : A) := v_2 \hookrightarrow \mu; \mathsf{blame}}$$

$$\text{VSTEP-U}\quad \frac{|u|_{\mu} = A}{\mu; u \hookrightarrow \mu; u : A}$$

$$\text{VSTEP-ANNO}\quad \frac{\neg \mathit{value}\ e : A \qquad \mu; e \hookrightarrow \mu'; e'}{\mu; e : A \hookrightarrow \mu'; e' : A}$$

Fig. 9: Reduction rules for λ^{G}_{gpr}.

rules VSTEP-BETAD, VSTEP-TAPD, VSTEP-DEREFP, and VSTEP-ASSIGND, we cast the value to the corresponding dynamic types and filter out programs with errors. To apply a value to a functional value (rules VSTEP-BETA and VSTEP-BETAP), the argument type must be consistent with function input types A_2. Moreover, the expected substituted value type is A_1. Thus, the argument value should be cast by A_2 and A_1, which may return a blame error. To preserve the type, the substituted body is annotated with B_1 and B_2. When a value v is annotated with a type A, the type of the value must be consistent with type A, and run-time checking is needed to validate consistency (rule VSTEP-ANNOV). A reference value ref v is bound in the store with a fresh location o (rule VSTEP-REFV). To obtain a value from the store by the location, from the last expression we use rule VSTEP-DEREF.

Note that in the typing rule for references:

$$\frac{\Sigma; \cdot \vdash o : A_1 \Rightarrow A_1 \qquad A_1 \rhd \mathsf{Ref}\, A}{\Sigma; \cdot \vdash !(o : A_1) \Rightarrow A} \textsc{Typ-deref}$$

The expected type is A but the bound value type is consistent with A. Thus we annotate v using type A. When assigning a value to replace the bound value in the reference using rules VSTEP-ASSIGN and VSTEP-ASSIGNP :

$$\frac{A \rhd \mathsf{Ref}\, A_2 \qquad \Sigma; \cdot \vdash o : A \Rightarrow A \qquad \Sigma; \cdot \vdash v_2 \Leftarrow A}{\Sigma; \cdot \vdash (o : A) := v_2 \Rightarrow \mathsf{Unit}} \textsc{Typ-assign}$$

The bound value by location o has type A_1, while the type of v_2 is consistent with type A_2 and A_2 is consistent with A_1. The expected type to be replaced is type A_1, therefore v_2 is cast by type A_1 and A_2. Note that the cast result can be blamed. If a type is applied to a polymorphic value, from the last expression (rule VSTEP-TAP):

$$\frac{B \rhd \forall X. B_2 \qquad \Sigma; \cdot \vdash (\Lambda X. e : A) : B \Rightarrow B}{\Sigma; \cdot \vdash ((\Lambda X. e : A) : B)\, C \Rightarrow B_2[X \mapsto C]} \textsc{Typ-tapp}$$

The expected type is $(B_2[X \mapsto C])$ but the substituted expression $(e[X \mapsto C] : A[X \mapsto C])$ has type $(A[X \mapsto C])$, so it is annotated with type $(B_2[X \mapsto C])$.

Properties of λ^G_{gpr}. λ^G_{gpr} is deterministic (Theorem 4) and type sound (Theorem 5 and Theorem 6).

Theorem 4 (Determinism of λ^G_{gpr}). *If $\Sigma; \cdot \vdash e \Leftrightarrow A$, $\mu; e \hookrightarrow \mu_1; r_1$ and $\mu; e \hookrightarrow \mu_2; r_2$ then $r_1 = r_2$ and $\mu_1 = \mu_2$.*

Theorem 5 (Type Preservation of λ^G_{gpr}). *If $\Sigma; \cdot \vdash e \Leftrightarrow A$, $\Sigma \vdash \mu$, and $\mu; e \hookrightarrow \mu'; e'$ then $\Sigma'; \cdot \vdash e' \Leftrightarrow A$, $\Sigma' \vdash \mu'$ and $\Sigma' \supseteq \Sigma$.*

Theorem 6 (Progress of λ^G_{gpr}). *If $\Sigma; \cdot \vdash e \Leftrightarrow A$ then e is a value or $\exists r\, \mu'$, $\mu; e \hookrightarrow \mu'; r$.*

4.3 Gradual Typing Criteria

Siek *et al.* [31,32] proposed a set of criteria for gradual typing system. At the end of the spectrum, a fully annotated gradually typed program should behave as a statically typed program. Conversely, a gradually typed program without annotations should behave as a dynamic program. Siek *et al.* proposed the gradual guarantee, which states that having annotations that are more/less precise should not change the behavior of the programs. Here we show that λ^G_{gpr} has the gradual guarantee.

To prove the gradual guarantee, we define the precision for types, expressions and stores. At the top of Figure 10 is type precision $A \sqsubseteq B$, which states that type A is more precise than B. The unknown type $\star$ is less precise than any other types. Each type is more precise than itself. The precision of functions, polymorphic functions and

$\boxed{A \sqsubseteq B}$ *(Type Precision)*

$$\text{TP-UNIT}\ \frac{}{\mathsf{Unit} \sqsubseteq \mathsf{Unit}} \qquad \text{TP-VAR}\ \frac{}{X \sqsubseteq X} \qquad \text{TP-Z}\ \frac{}{\mathsf{Int} \sqsubseteq \mathsf{Int}} \qquad \text{TP-DYN}\ \frac{}{A \sqsubseteq \star} \qquad \text{TP-ARR}\ \frac{A_1 \sqsubseteq B_1 \qquad A_2 \sqsubseteq B_2}{A_1 \to A_2 \sqsubseteq B_1 \to B_2}$$

$$\text{TP-FORALL}\ \frac{A \sqsubseteq B}{\forall X.\, A \sqsubseteq \forall X.\, B} \qquad \text{TP-REF}\ \frac{A \sqsubseteq B}{\mathsf{Ref}\, A \sqsubseteq \mathsf{Ref}\, B}$$

$\boxed{e_1 \sqsubseteq e_2}$ *(Expression Precision)*

$$\text{EP-LIT}\ \frac{}{i \sqsubseteq i} \qquad \text{EP-VAR}\ \frac{}{x \sqsubseteq x} \qquad \text{EP-UNIT}\ \frac{}{\mathsf{unit} \sqsubseteq \mathsf{unit}} \qquad \text{EP-O}\ \frac{}{o \sqsubseteq o} \qquad \text{EP-REF}\ \frac{e_1 \sqsubseteq e_2}{\mathsf{ref}\, e_1 \sqsubseteq \mathsf{ref}\, e_2} \qquad \text{EP-DEREF}\ \frac{e_1 \sqsubseteq e_2}{!e_1 \sqsubseteq !e_2}$$

$$\text{EP-ABS}\ \frac{\begin{array}{c} e_1 \sqsubseteq e_2 \\ A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq B_2 \end{array}}{\lambda x.\, e_1 : A_1 \to B_1 \sqsubseteq \lambda x.\, e_2 : A_2 \to B_2} \qquad \text{EP-APP}\ \frac{e_1 \sqsubseteq e_3 \qquad e_2 \sqsubseteq e_4}{e_1\, e_2 \sqsubseteq e_3\, e_4} \qquad \text{EP-ASSIGN}\ \frac{e_1 \sqsubseteq e_3 \qquad e_2 \sqsubseteq e_4}{e_1 := e_2 \sqsubseteq e_3 := e_4}$$

$$\text{EP-ANNO}\ \frac{e_1 \sqsubseteq e_2 \qquad A_1 \sqsubseteq A_2}{e_1 : A_1 \sqsubseteq e_2 : A_2} \qquad \text{EP-TABS}\ \frac{e_1 \sqsubseteq e_2 \qquad A_1 \sqsubseteq A_2}{\Lambda X.\, e_1 : A_1 \sqsubseteq \Lambda X.\, e_2 : A_2} \qquad \text{EP-TAPP}\ \frac{e_1 \sqsubseteq e_2 \qquad A_1 \sqsubseteq A_2}{e_1\, A_1 \sqsubseteq e_2\, A_2}$$

$\boxed{\mu_1 \sqsubseteq \mu_2}$ *(Store Precision)*

$$\text{SP-NIL}\ \frac{}{\cdot \sqsubseteq \cdot} \qquad \text{SP-EMPTY}\ \frac{\mu_1 \sqsubseteq \mu_2 \qquad v_1 \sqsubseteq v_2}{\mu_1, o = v_1 \sqsubseteq \mu_2, o = v_2}$$

Fig. 10: Precision Relation.

reference types holds, if the precision of their sub-components holds. Note that the precision of function types is "covariant" in the argument types since to compare the precision of the two programs:

$$\lambda x.\, 1 : \mathsf{Int} \to \mathsf{Int}$$
$$\lambda x.\, 1 : \star \to \mathsf{Int}$$

we should just say that the first one is more precise than the second one because the input type of the second one is fully dynamic. Expression precision is shown in the middle of Figure 10. The rules can mostly be derived from the type precision. Each expression is in a precision relation with itself. Structural expressions are in a precision relation if their sub-expressions are related. Lastly, store precision, shown at the bottom of Figure 10, shows that precision holds if the precision of values in the store holds.

$\boxed{\mu; e \hookrightarrow_{s*} \mu'; e'}$ *(Operational semantics)*

$$\textsc{Step-eval}\quad \frac{\mu; e \hookrightarrow_{s*} \mu'; e'}{\mu; F[e] \hookrightarrow_{s*} \mu'; F[e']}
\qquad
\textsc{Step-annov}\quad \frac{}{\mu; u : A : A \hookrightarrow_{s*} \mu; u : A}
\qquad
\textsc{Step-assign}\quad \frac{}{\mu; o := v \hookrightarrow_{s*} \mu[o \mapsto v]; \mathsf{unit}}$$

$$\textsc{Step-tap}\quad \frac{}{\mu; ((\Lambda X.\, e : A) : \forall X.\, A)\; A \hookrightarrow_{s*} \mu; e[X \mapsto A] : A[X \mapsto A]}
\qquad
\textsc{Step-deref}\quad \frac{o = v\, \in \mu}{\mu; !o \hookrightarrow_{s*} \mu; v : A}$$

$$\textsc{Step-beta}\quad \frac{}{\mu; ((\lambda x.\, e : A \to B) : A \to B)\; v \hookrightarrow_{s*} \mu; e[x \mapsto v] : B : B}
\qquad
\textsc{Step-refv}\quad \frac{o \notin \mu}{\mu; \mathsf{ref}\; v \hookrightarrow_{s*} \mu, o = v; o}$$

$$\textsc{Step-u}\quad \frac{|u|_\mu = A}{\mu; u \hookrightarrow_{s*} \mu; u : A}
\qquad
\textsc{Step-anno}\quad \frac{\neg \mathit{value}\; e : A \qquad \mu; e \hookrightarrow_{s*} \mu'; e'}{\mu; e : A \hookrightarrow_{s*} \mu'; e' : A}$$

Fig. 11: Reduction rules for λ_{gpr}.

Static criteria. We show that the full static type system of λ^G_{gpr} is equivalent to the λ_{gpr} calculus (Theorem 7). We use s to denote a relation from the static system in case of ambiguity. Theorem 8 shows the static gradual guarantee of λ^G_{gpr}. If a more precise program is well-typed then a less precise program should be well-typed with a less precise type.

Theorem 7 (Equivalence for λ_{gpr} (statics)). *If $\cdot; \cdot \vDash_s e \;\Leftrightarrow\; A$ if and only if $\cdot; \cdot \vdash e \;\Leftrightarrow\; A$.*

Theorem 8 (Static Gradual Guarantee). *If $e_1 \sqsubseteq e_2$, $\cdot; \cdot \vdash e_1 \;\Leftrightarrow\; A$ then $\cdot; \cdot \vdash e_2 \;\Leftrightarrow\; B$ and $A \sqsubseteq B$.*

Dynamic criteria. Theorem 9 says that fully static programs of λ^G_{gpr} calculus behaves in the same as the λ_{gpr} at run-time. To make the proofs easier, the reduction rules of λ_{gpr} calculus have extra annotations to follow λ^G_{gpr} (we denoted as $s*$). It means that there are extra identical annotations, as shown in the gray parts of Figure 4. However, these annotations are identical and they can be removed without affecting the final reduction result. In addition, as in λ^G_{gpr}: values have annotations; raw values should step to be annotated values; and annotations are not included in Frames. This requires a few extra rules, which are shown in Figure 11.

Notably, λ^G_{gpr} has the dynamic gradual guarantee (Theorem 10). The proof is simple in comparison to the original proof by Siek et al. [32]. This simple theorem is formalized following the work of Garcia *et al.* [12]. It says that if a more precise program with a more precise store can reduce, then the less precise program with a less precise store can also reduce. Furthermore, their resulting programs and stores should keep the precision relation.

Theorem 9 (Equivalence for λ_{gpr} (dynamic)). *$\forall \cdot;\cdot \vDash_s e \Leftrightarrow A$,*

- *If $\mu; e \hookrightarrow_{s*} \mu'; e'$ then $\mu; e \hookrightarrow \mu'; e'$.*
- *If $\mu; e \hookrightarrow \mu'; e'$ then $\mu; e \hookrightarrow_{s*} \mu'; e'$.*

Theorem 10 (Dynamic Gradual Guarantee). *If $e_1 \sqsubseteq e_2$, $\mu_1 \sqsubseteq \mu_2$, $\cdot;\cdot \vdash e_1 \Leftrightarrow A$, $\cdot;\cdot \vdash e_2 \Leftrightarrow B$ and $\mu_1; e_1 \hookrightarrow \mu_1'; e_1'$ then there exists e_2' and μ_2' such that $\mu_2; e_2 \hookrightarrow \mu_2'; e_2'$, $e_1' \sqsubseteq e_2'$ and $\mu_1' \sqsubseteq \mu_2'$.*

5 Discussion

In this section, we briefly discuss alternative designs and possible extensions.

Preserving relational parametricity. An alternative design is to have a directed semantics gradual polymorphism calculi, which preserves parametricity. We employ the eager semantics similar to the AGT methodology, which is applied in the GSF calculus. Toro *et al.* [37] analyzed the following example to show how parametricity is broken by the naive use of the dynamic sealing in the eager semantics:

$$(\Lambda X.(\lambda x : X.\mathsf{let}\ y : \star = x\ \mathsf{in}\ \mathsf{let}\ z : \star = y\ \mathsf{in}\ z + 1))\ \mathsf{Int}\ 1$$

The polymorphic function with type $(\forall X.X \to \star)$ breaks parametricity, which should be detected at run-time and raise an error. However, the application of the function reduces to 2. A fresh name variable α is generated and is bounded to the type Int. Variable x to y is flowing from type Int to type α; y to z is flowing from type $\star$ to type $\star$; and x to z is flowing from Int to $\star$. Any of these type flows are safe. Thus the reason for the loss of parametricity is related to the loss of precise type information. Consequently, dynamic sealing is not enough to enforce relational parametricity. For the above example, GSF detects the error by the refining evidences such as $(\langle \alpha^{E_1}, \alpha^{E_2} \rangle)$. Importantly in the type flow from y to z, more precise types (Int and α^{Int}) instead of $\star$ and $\star$ are obtained, so when moving from x to z the type changes from Int to α^{Int}. When doing the addition, the run-time error is detected since the flow from α^{Int} to Int is not defined. A potential approach for us is to use tracked types ($A^{<B_1,B_2>}$), which are similar to the refined evidences in the GSF calculus. Because λ^{G}_{gpr} is a source language, we do not have evidences, thus a possible approach is to record information in types. For the above example, tracked types can track the unknown type with more precise types from y to z to be Int and α^{Int} which is $\star^{(\mathsf{Int},\alpha^{Int})}$ and then from x to z to be $\star^{(\mathsf{Int},\alpha^{Int})}$ as the refined evidences and a run-time error is detected when doing the addition.

A space-efficient gradual polymorphic calculus. Ozaki *et al.* [27] explored the space efficiency problem in the gradual polymorphic calculus. They extended the coercion calculus (λC) [29] with parametric polymorphism (called $\lambda C^{\forall}$). Dynamic sealing was applied in $\lambda C^{\forall}$ to enforce relational parametricity. Consequently, a sequence of coercions is allowed and they showed that it cannot be normalized to a smaller coercion. In other words, the size of sequences is unbounded. Notably, they stated and proved that $\lambda C^{\forall}$ cannot be space-efficient when dynamic sealing is supported. Furthermore, they conjectured that the gradual polymorphic calculus with dynamic sealing cannot become space-efficient. Our λ^{G}_{gpr} calculus substitutes types directly, as the traditional semantics without employing dynamic sealing. Moreover, the eager semantics is applied. Thus we believe that it is possible for our λ^{G}_{gpr} calculus to be a space-efficient gradual polymorphic calculus. Two tentative and promising rules are as follows:

$$\frac{A \sim C}{e : A : B : C \hookrightarrow e : A : C} \qquad \frac{\neg A \sim C}{e : A : B : C \hookrightarrow \mathsf{blame}}$$

With the above two rules, annotations are removed or an error is raised, to achieve the space-efficient goal. Surprisingly, with these two rules, it seems possible to have a space-efficient gradual references calculus naturally. We intend to explore this in the future.

Implicit polymorphic references. Implicit (higher-rank) polymorphism [10,26,19] is pervasive in theoretic and practical programming languages. Existing gradual polymorphic calculi are mainly explicitly polymorphic. One exception is the work of Xie *et al.* [41]. Explicit polymorphism means that polymorphic types are not related to any of its instantiated types but in implicit polymorphism, they are related. Xie *et al.* [41] designed a source gradual implicit polymorphism calculus with consistent subtyping but their dynamic semantics is defined by translating to the well-known polymorphic blame calculus ($\lambda B^{\forall}$) [3] without the proof of the dynamic gradual guarantee. A possible extension of Xie et al.'s work is to support implicit polymorphism with a direct dynamic semantics, and to explore the dynamic gradual guarantee and parametricity properties. However, it is well-known that a naive combination of implicit polymorphism and references lead to an unsound language. A possible solution is to limit polymorphism to syntactic let-bound values as adopted by Standard ML [40].

Alternative forms of values. In our calculus, all values are annotated, such as $1 : \mathsf{Int}$ or $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int}$. This introduces some overhead as some annotations are redundant. We can have an alternative and workable form of values as follows:

$$v ::= u \mid u : \star \mid (\Lambda X.\, e : A) : \forall X.\, B \mid (\lambda x.\, e : A_1 \to B_1) : A_2 \to B_2$$

The above value form removes redundant annotations such as integers ($1 : \mathsf{Int}$). This is good for performance, but it would make the proof of dynamic gradual guarantee harder. However, the resulting calculus with fewer annotations should have an equivalent semantics to our calculus, and would be a better candidate for guiding an implementation.

6 Related Work

Gradual typing. Gradual typing is a term coined by Siek *et al.* [31]. The unknown type ?, which we represent as $\star$, is the new notion introduced to a gradual type system to integrate dynamic and static typing. By using the unknown type $\star$, equality on types is lifted to consistency. Any type is consistent with type $\star$. Therefore, run-time type checking is needed for a gradually typed lambda calculus. Traditionally, the dynamic semantics of a gradual language is defined by elaborating to a target language, which includes cast calculi [39,34,29,11,3] and coercion calculi [13,14,30,29,27].

Garcia *et al.* [12] proposed the abstracting gradual typing (AGT) approach, which allows for deriving a gradual type system by lifting the static type system. They argue about the weakness of elaborating to a target language, and did not resort to a target language in their calculus by using intrinsic terms. Our λ_{gpr}^{G} defines the dynamic semantics directly without using intrinsic terms, but employing instead an approach based on type-directed operational semantics (TDOS). Type directed operational semantics (TDOS) was proposed by Huang *et al.* [15] to design calculi with the merge operator and intersection types. Ye *et al.* [42] explored the use of the TDOS in gradual typing. In TDOS, type annotations are relevant at runtime and can affect the semantics, unlike many traditional calculi where types are not runtime relevant. With a TDOS we can design a gradually typed calculus without elaboration to a cast calculus, since the semantics can be given directly. Our λ_{gpr}^{G} employs the eager semantics for higher-order values following an approach similar to AGT. Ye *et al.* only consider a TDOS for a simply typed, purely functional language. Our work shows that the TDOS approach can be extended to important features, such as polymorphism and references.

Gradual typing with references. Many languages with static and dynamic typing, employing some form of optional typing, support references. These include Flow [8], Dart [6] and TypeScript [5]. However for optional typing, the run-time checking is not performed for fully dynamic programs, leading to unsoundness with respect to the static type system. In the work of Siek *et al.* [31], he already considered mutable references, but in a very simple setting without annotation expressions. Furthermore, the gradually typed lambda calculus is elaborated to a target language to define the dynamic semantics. Herman *et al.* [14] designed a coercion calculus with references, which is space efficient. A gradualizer, introduced by Cimini and Siek [9], can derive a gradual static type system and cast insertion with references systematically. Toro *et al.* [38] designed source gradual typing system with references $\lambda_{\widetilde{REF}}$ and a corresponding target language $\lambda_{\widetilde{REF}}^{\epsilon}$ using the Abstracting Gradual Typing (AGT) methodology. They designed the $\lambda_{\widetilde{REF}}^{\epsilon}$ as a space-efficient calculus and proved the gradual guarantee. Our λ_{gpr}^{G} is the first polymorphic gradually typed language with references.

Existing gradual polymorphic calculi. In the following we summarize some of the solutions to the problem of preserving parametricity and gradual guarantee in gradual polymorphic calculi and the changes that these solutions entail.

Dynamic sealing. Ahmed et al. [3] solved the problem in Section 2 by using dynamic sealing, inspired by the work of Matthews *et al.* [21]. They proposed the polymorphic blame calculus [3] (we present it as $\lambda B^{\forall}$), which is a widely used cast calculus with dynamic sealing. The most interesting construct of $\lambda B^{\forall}$ is the named type binding $\nu X := A.t$, which is introduced to record the instantiated type of a type variable. The programs in Section 2 behave as expected in $\lambda B^{\forall}$:

$$\begin{aligned}
&(K^{\star} : \star \Rightarrow \forall X. \forall Y. X \to Y \to X)\,\mathsf{Int}\,\mathsf{Int}\,2\,3 \\
\hookrightarrow^{*}\ &\nu Y := \mathsf{Int}.\nu X := \mathsf{Int}.(2 : X \Rightarrow \star : \star \Rightarrow X) \\
\hookrightarrow^{*}\ &2 \\
&(K^{\star} : \star \Rightarrow \forall X. \forall Y. X \to Y \to Y)\,\mathsf{Int}\,\mathsf{Int}\,2\,3 \\
\hookrightarrow^{*}\ &\nu Y := \mathsf{Int}.\nu X := \mathsf{Int}.(2 : X \Rightarrow \star : \star \Rightarrow Y) \\
\hookrightarrow^{*}\ &\mathit{blame}
\end{aligned}$$

The first program succeeds and returns the first argument. While the second program fails, since the polymorphic information is recorded as $X := \mathsf{Int}$ and $Y := \mathsf{Int}$ in type bindings and the original type variable names are preserved in the casts. Notably, for higher-order values, $\lambda B^{\forall}$ follows the lazy semantics as the blame calculus [39,29]. That is, for a function value, the checking is delayed until an argument value is applied. This, unfortunately results in unbounded space consumption for higher-order casts [13,14].

As Xie *et al.* [41] pointed out, the compatibility relation of $\lambda B^{\forall}$ mixes explicit and implicit polymorphism to some extent, since they employ the following rule:

$$\frac{A[X \mapsto \star] \prec B}{\forall X. A \prec B}$$

This compatibility rule of $\lambda B^{\forall}$ allows $\forall X. X \to X$ to be compatible with any static instantiated types such as $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Bool}$. These types are not related in System F so $\lambda B^{\forall}$ is not a conservative extension of System F. The gradual guarantee has not been discussed in $\lambda B^{\forall}$, but they show the parametricity property.

The F_G and F_C calculi. Igarashi *et al.* [17] improved on $\lambda B^{\forall}$. They designed a source calculus (F_G) and a target calculus (F_C), which is a conservative extension of System F. The dynamic semantics of F_G is indirect and defined by translation to F_C. F_G does not relate $\forall X. X \to X$ with static instantiations, but only with the dynamic instantiation $\star \to \star$. The type $\star \to \star$ is called quasi-polymorphic, since it is an instantiation of $\forall X. X \to X$ similarly to what happens with implicit polymorphism. However, a type such as $\mathsf{Int} \to \mathsf{Int}$ is not quasi-polymorphic. Instead of binding types locally by ($\nu X := A.t$), they made the type bindings global. Their reduction form $\Sigma \triangleright f \hookrightarrow \Sigma' \triangleright f'$ is augmented with a store, which records the bounded type variables $X := A$. The above example reduces in F_C as follows.

$$\begin{aligned}
&\Sigma \triangleright (K^{\star} : \star \Rightarrow \forall X. \forall Y. X \to Y \to X)\ \mathsf{Int}\ \mathsf{Int}\ 2\ 3 \\
\hookrightarrow^{*}\ &\Sigma \triangleright (\Lambda X.\Lambda Y.K^{\star} : \star \Rightarrow X \to Y \to X)\ \mathsf{Int}\ \mathsf{Int}\ 2\ 3 \\
\hookrightarrow^{*}\ &\Sigma, X := \mathsf{Int}, Y := \mathsf{Int} \triangleright (K^{\star} : \star \Rightarrow X \to Y \to X)\ \mathsf{Int}\ 2\ 3 \\
\hookrightarrow^{*}\ &2
\end{aligned}$$

Furthermore, they argue that type bindings generated locally lead to run-time overheads. Their observation is that type bindings are not required for every substitution, but only for casts with the dynamic type ($\star$). Therefore they employ two kinds of type variables, which are distinguished by labels. One kind is static type variables (X::S) and the other kind is gradual type variables (X::G). Type application for static type abstraction does not generate type bindings, which are only generated for gradual type abstractions. Parametricity and the static gradual guarantee are proved, although the proofs are not mechanized. However, the dynamic gradual guarantee is left as conjecture. In addition their static gradual guarantee is proved with some constraints in the type precision relation. In their precision, $\forall X. X \to X$ is more precise than $\forall X. X \to \star$ but not $\forall X. \star \to X$.

The GSF calculus. Toro *et al.* [37] presented the gradual polymorphic calculus (named GSF), which employs the Abstracting Gradual Typing (AGT) methodology. In AGT, casting of higher-order values is eager compared to $\lambda B^{\forall}$ and F_C. This avoids the problem of space consumption although, as New *et al.* [25] pointed out, the η principle (which ensures $V \equiv \lambda x.Vx$ in the call-by-value languages) is broken. To preserve parametricity, global dynamic sealing, which does not distinguish between static and gradual variables, is used. They also refine the presentation of evidence, which witnesses the consistency judgement, ensuring that it holds. Instead of simple evidences such as ($\langle \alpha, Int \rangle$), they employ sealing evidences ($\langle \alpha^E, Int \rangle$). GSF satisfies parametricity but not the gradual guarantee. Importantly, they proved that the gradual guarantee is incompatible with parametricity.

Parametricity with the Gradual Guarantee. To achieve both parametricity and the gradual guarantee, New *et al.* [24] designed $PolyG^{\nu}$ calculus which gave up the syntax of System F and the users are required to provide different sealing options. They introduced the sealed syntax as $seal_X\ M$ which explicitly seals terms. With the user-defined syntax, the gradual guarantee and parametricity are proved. More recently, Labrada *et al.* [20] improve on GSF. They do not change the syntax of System F but insert plausible sealing forms during the elaboration from a gradual source language which is named Funk to a target cast calculus. They proved the gradual guarantee and parametricity for the target language, but for the source language (Funk), the gradual guarantee comes with a restriction for type applications, which can only be instantiated with base and variable types. Some of the main theorems are proved in Agda.

Summary. In order to keep parametricity we need several compromises. For instance, we need to use a dynamic sealing mechanism instead of direct type substitution causing extra space and time consumption. In many of the earlier calculi, the gradual guarantee is not obtained. In the later calculi, the gradual guarantee is either restricted or we need to give up the syntax of System F. Traditionally, many works on gradual typing are based on two different calculi: a source gradually typed language, and a target cast/-coercion calculus where casts/coercions are explicit. The dynamic semantics is defined by elaborating the source language to the target calculus. In other words, the semantics of the gradually typed language is given indirectly via a second, target language. All previously discussed works follow this indirect way to give the semantics to a gradually typed source language.

Furthermore, none of the gradually typed polymorphic calculi supports references. However, even for a static polymorphic calculus extended with mutable references ob-

	$\lambda B^{\forall}$	F_G	GSF	$PolyG^{\nu}$	Funk	λ^{G}_{gpr}
	2011	2017	2019	2020	2022	present work
Direct Substitution	×	×	×	×	×	✓
System F extension	×	✓	✓	×	✓	✓
Direct Semantics	×	×	×	×	×	✓
Parametricity	✓	✓	✓	✓	✓	×
Gradual Guarantee	×	×	×	✓	✓́	✓
Semantics	Lazy	Lazy	Eager	Lazy	Eager	Eager
Mechanized Proofs	×	×	×	×	✓́	✓
References	×	×	×	×	×	✓

Table 1: Comparison among gradual polymorphism calculi. A × denotes no. A ✓ denotes yes while ✓́ denotes partial yes.

taining parametricity is highly non-trivial. As Ahmed *et al.* [2] stated: "*combing mutable references with polymorphism can be extremely tricky*." From the analysis of Jaber and Tzevelekos [18], we know that naively moving from a polymorphic calculus to incorporate with mutable references, breaks parametricity. The reason is that common references can be instantiated with differently typed variables. Therefore, extending a gradual polymorphic calculus with the mutable references is non-trivial, and none of the existing gradual languages with polymorphism support references.

Table 1 summarizes several features and differences in existing gradually polymorphic calculi.

7 Conclusion

In this paper, we design a static system λ_{gpr} with polymorphism and references and its gradual counterpart λ^{G}_{gpr}. λ^{G}_{gpr} has a direct semantics without resorting to a cast calculi. In λ^{G}_{gpr}, the gradual guarantee is proved but we give up parametricity. In exchange, our calculus can be simplified, since sophisticated mechanisms such as dynamic sealing are not needed. Our calculus follows the original semantics of System F, based on direct type substitutions, avoiding extra space and time complexity that is necessary by mechanisms such as dynamic sealing. In the future, we could try to find out if there is a way to keep both gradual guarantee and relational parametricity for the source language, or explore more efficient formulations of λ^{G}_{gpr}.

Acknowledgements We are grateful to anonymous reviewers and our colleagues at the HKU PL group. This work has been sponsored by Hong Kong Research Grants Council projects number 17209520 and 17209821.

References

1. Abadi, M., Cardelli, L., Pierce, B.C., Plotkin, G.D.: Dynamic typing in a statically-typed language. In: POPL '89 (1989)
2. Ahmed, A., Appel, A.W., Virga, R.: An indexed model of impredicative polymorphism and mutable references (2003)
3. Ahmed, A., Findler, R.B., Siek, J.G., Wadler, P.: Blame for all. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 201–214 (2011)
4. Barendregt, H.P., Church, A.: The impact of the lambda calculus (2014)
5. Bierman, G., Abadi, M., Torgersen, M.: Understanding typescript. In: European Conference on Object-Oriented Programming. pp. 257–281. Springer (2014)
6. Bracha, G.: The dart programming language. Addison-Wesley Professional (2015)
7. Cartwright, R., Fagan, M.: Soft typing. In: PLDI '91 (1991)
8. Chaudhuri, A.: Flow: a static type checker for javascript. SPLASH-I In Systems, Programming, Languages and Applications: Software for Humanity (2015)
9. Cimini, M., Siek, J.G.: The gradualizer: a methodology and algorithm for generating gradual type systems. Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2016)
10. Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, pp. 429–442. ICFP '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2500365.2500582, https://doi.org/10.1145/2500365.2500582
11. Garcia, R.: Calculating threesomes, with blame. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming. pp. 417–428 (2013)
12. Garcia, R., Clark, A.M., Tanter, E.: Abstracting gradual typing. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 429–442. POPL '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837670, https://doi.org/10.1145/2837614.2837670
13. Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. In: In Trends in Functional Programming (TFP (2007)
14. Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. Higher-Order and Symbolic Computation **23**(2), 167 (2010)
15. Huang, X., Oliveira, B.C.d.S.: A Type-Directed Operational Semantics For a Calculus with a Merge Operator. In: Hirschfeld, R., Pape, T. (eds.) 34th European Conference on Object-Oriented Programming (ECOOP 2020). Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, pp. 26:1–26:32. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.26, https://drops.dagstuhl.de/opus/volltexte/2020/13183
16. Huang, X., Zhao, J., Oliveira, B.C.d.S.: Taming the merge operator. Journal of Functional Programming **31**, e28 (2021)
17. Igarashi, Y., Sekiyama, T., Igarashi, A.: On polymorphic gradual typing. Proc. ACM Program. Lang. **1**(ICFP) (aug 2017). https://doi.org/10.1145/3110284, https://doi.org/10.1145/3110284
18. Jaber, G., Tzevelekos, N.: Trace semantics for polymorphic references*. 2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) pp. 1–10 (2016)
19. Jones, S.L.P., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. Journal of Functional Programming **17**, 1 – 82 (2007)

20. Labrada, E., Toro, M., Tanter, E., Devriese, D.: Plausible sealing for gradual parametricity. Proc. ACM Program. Lang. **6**(OOPSLA1) (apr 2022). https://doi.org/10.1145/3527314, `https://doi.org/10.1145/3527314`
21. Matthews, J., Ahmed, A.: Parametric polymorphism through run-time sealing or, theorems for low, low prices! In: Drossopoulou, S. (ed.) Programming Languages and Systems. pp. 16–31 (2008)
22. Matthews, J., Findler, R.B.: Operational semantics for multi-language programs. ACM Trans. Program. Lang. Syst. **31**(3) (apr 2009). https://doi.org/10.1145/1498926.1498930, `https://doi.org/10.1145/1498926.1498930`
23. Møgelberg, R.E., Simpson, A.K.: Relational parametricity for computational effects. Logical Methods in Computer Science **5**, 1–31 (2009)
24. New, M.S., Jamner, D., Ahmed, A.: Graduality and parametricity: together again for the first time. Proceedings of the ACM on Programming Languages **4**, 1 – 32 (2020)
25. New, M.S., Licata, D.R., Ahmed, A.: Gradual type theory. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290328, `https://doi.org/10.1145/3290328`
26. Odersky, M., Läufer, K.: Putting type annotations to work. In: POPL '96 (1996)
27. Ozaki, S., Sekiyama, T., Igarashi, A.: Is space-efficient polymorphic gradual typing possible? (2021)
28. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress (1983)
29. Siek, J., Thiemann, P., Wadler, P.: Blame and coercion: Together again for the first time. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 425–435. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737968, `https://doi.org/10.1145/2737924.2737968`
30. Siek, J.G., Garcia, R., Taha, W.: Exploring the design space of higher-order casts. In: ESOP (2009)
31. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop. vol. 6, pp. 81–92 (2006)
32. Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined criteria for gradual typing. In: 1st Summit on Advances in Programming Languages (SNAPL 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
33. Siek, J.G., Vitousek, M.M., Cimini, M., Tobin-Hochstadt, S., Garcia, R.: Monotonic references for efficient gradual typing. In: ESOP (2015)
34. Siek, J.G., Wadler, P.: Threesomes, with and without blame. In: Proceedings for the 1st Workshop on Script to Program Evolution. p. 34–46. STOP '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1570506.1570511, `https://doi.org/10.1145/1570506.1570511`
35. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: from scripts to programs. In: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. pp. 964–974 (2006)
36. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: POPL '08 (2008)
37. Toro, M., Labrada, E., Tanter, É.: Gradual parametricity, revisited. Proceedings of the ACM on Programming Languages **3**, 1 – 30 (2019)
38. Toro, M., Tanter, É.: Abstracting gradual references. Sci. Comput. Program. **197**, 102496 (2020)
39. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: European Symposium on Programming. pp. 1–16. Springer (2009)
40. Wright, A.K.: Simple imperative polymorphism. LISP and Symbolic Computation **8**, 343–355 (1995)

41. Xie, N., Bi, X., d. S. Oliveira, B.C.: Consistent subtyping for all. ACM Transactions on Programming Languages and Systems (TOPLAS) **42**, 1 – 79 (2018)
42. Ye, W., Oliveira, B.C.d.S., Huang, X.: Type-directed operational semantics for gradual typing. In: 35th European Conference on Object-Oriented Programming (ECOOP 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2021)

Modal Crash Types for Intermittent Computing*

Farzaneh Derakhshan(✉), Myra Dotzel, Milijana Surbatovich, and Limin Jia

Carnegie Mellon University, Pittsburgh PA, USA
{fderakhs,mdotzel,milijans,liminjia}@andrew.cmu.edu

Abstract. Intermittent computing is gaining traction in application domains such as Energy Harvesting Devices (EHDs) that experience arbitrary power failures during program execution. To make progress, programs require system support to checkpoint state and re-execute after power failure by restoring the last saved state. This re-execution should be *correct*, i.e., simulated by a continuously-powered execution. We study the logical underpinning of intermittent computing and model checkpoint, crash, restore, and re-execution operations as computation on Crash types. We draw inspiration from adjoint logic and define Crash types by introducing two adjoint modality operators to model persistent and transient memory values of partial (re-)executions and the transitions between them caused by checkpoints and restoration. We define a Crash type system for a core calculus. We prove the correctness of intermittent systems by defining a novel logical relation for Crash types.

Keywords: intermittent computing · modal Crash type · logical relation

1 Introduction

Intermittent computing is gaining importance in application domains that require inaccessible or large-scale device deployments, such as wildlife monitoring [28], tiny satellites [22,29], or smart civil infrastructure [1]. As battery maintenance may be infeasible in these environments, programs can instead run on batteryless Energy Harvesting Devices (EHDs). An EHD can run solely off energy harvested from its environment, at the cost of being powered intermittently. The device harvests energy (e.g., via solar panel) into a re-chargeable buffer. Once the energy buffer is full, the device turns on and begin to compute, consuming the stored energy. When the buffer drains, the device turns off at an arbitrary location until it can recharge and repeat this operational cycle. A power failure erases volatile execution state (e.g., the program counter), while

* This work was generously funded in part through National Science Foundation (NSF) Award 2007998, NSF Graduate Research Fellowship Program grants DGE1745016 and DGE2140739, and the CMU CyLab Security & Privacy Institute. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsoring organizations.

nonvolatile state persists. For programs to make progress, they require *intermittent system* support to save state at checkpoints and restore the saved state after power failure, potentially causing re-execution from the last checkpoint.

As EHDs aim to enable long-term deployments with little or no maintenance, intermittent systems must execute programs reliably despite frequent power failures and partial executions. Initial systems [35,43,24] relied only on informal notions of correctness that left them susceptible to memory consistency bugs caused by reading the results of partial executions [23] or by allowing sensor reads from past executions to remain in the nonvolatile memory [39]. More recent work [41,40,9,13] provides formal frameworks and correctness criteria for reasoning about intermittent execution. More concretely, all intermittent executions of a program must be simulated by some continuously-powered execution [41]. In other words, intermittent execution should be *idempotent*. Even if the system induces multiple partial executions of a program due to power failure, the program should not generate a different result than it would on a single execution.

The correctness of an intermittent execution relies on checkpointing, restoring, and finalizing state upon reaching the next checkpoint; mistakes in these operations can lead to incorrect, non-idempotent behavior. Few works have tried to understand the *fundamental logical underpinning* of these operations. This work fills this gap by formalizing checkpointing, crash, restoration, and re-execution as computation on *Crash types*. Crash types capture the core notion of intermittent computing: some values and computations persist across power failures and others do not. For instance, nonvolatile memory state persists across power failure and reboots, while volatile memory does not. Conversely, partially computed results do (or rather *should*) not persist across power failures, while completed/checkpointed computations do. We call the former *unstable* values and computations and the latter *stable* values and computations. Our key insight is that the interactions between these stable and unstable components bear close resemblance to shifts in adjoint logic [8,36]. Computation of a stable value can only rely on locations that store stable values, while computation on unstable values can rely on both stable and unstable values. Moreover, checkpoint and restore operations can turn values of one type to the other. We define terms and their associated types so that each of the key intermittent computing operations must be well-typed under our Crash types.

We define a core calculus for intermittent computing and develop a type system for Crash types by using the two adjoint modality operators. The Crash type of an intermittent computation is: $\mathtt{C}_{\mathtt{unit}} = {\downarrow}(\mathtt{nat} \rightsquigarrow {\uparrow}\mathtt{C}_{\mathtt{unit}}) \vee {\downarrow}{\uparrow}\mathtt{unit}$, which says that the computation will either encounter a power failure (the left disjunct), or succeed in producing a stable value (the right disjunct). In the former case, the computation is suspended until energy arrives, after which it will again act as an intermittent computation. This recursive definition captures the multiple re-executions of a computation under repeated power failures. To prove the correctness of intermittent systems, we define a novel logical relation for Crash types, indexed by the number of power failures, which relates a continuously-powered execution to an intermittent execution. While intermittent computing motivates our results, the methods we develop are generally applicable to other system failures with the same effect on persistent and transient storage.

This paper makes the following technical contributions:

- The first logical interpretation of key operations of intermittent execution.
- Novel Crash types to specify how stable and unstable portions of the system and computation interact.
- A core calculus for Crash types with progress and preservation.
- A novel logical relation to prove the correctness of intermittent executions.

Detailed proofs and definitions can be found in the extended TR [15].

2 Background

We provide background on intermittent computing and detail how checkpoint systems work to store and restore program state to handle power failures.

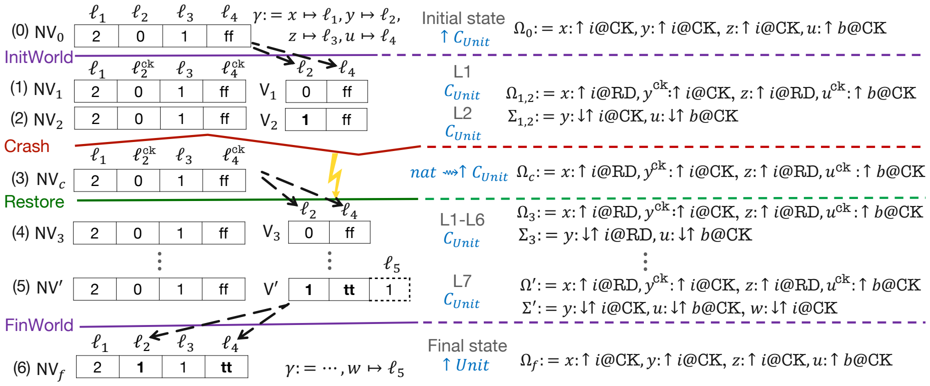
Intermittent Computing on EHDs. EHDs need intermittent system support to save necessary state before power failure and to restore it after reboot. When and where such checkpoints occur governs the *intermittent execution model* under which software executes. The two prevailing intermittent execution models are just-in-time (JIT) checkpoints [5,4] and atomic execution [23,24,43,37]. Under a JIT model, state is saved immediately prior to power failure so that execution resumes from the same point after reboot. Under an atomic execution model, state is saved at the beginning of an *atomic region*. If power fails before the end of the region, the system will reboot to the beginning of the region, re-executing until the region completes without power failure (akin to software transactions [38]). State-of-the-art intermittent systems use a hybrid "JIT + Atomics" model that defaults to JIT checkpoints except when there is an explicit atomic region [40,25,19]. Our core calculus follows this hybrid model.

To ensure idempotence, an intermittent system must save the value of volatile state and often a portion of the *nonvolatile* state. To illustrate why, consider an execution of the simple program in Fig. 1. The program has four variables stored in nonvolatile memory: x, y, and z of type `int` and u of type `bool`. It consists of two code blocks: an atomic region declared with the `Ckpt` construct (lines 1-7 on the left of Fig. 1) and a regular code block executed in JIT mode (lines 8-14 on the right). A continuous execution of the atomic region with initial state $x = 2, y = 0, z = 1, u = \mathtt{ff}$ ends in $x = 2, y = 1, z = 1, u = \mathtt{tt}$. Now, suppose power fails after the execution of Line 2. Once the device recharges, the program restarts from the start of the atomic region. If the system does not restore y's original value, this re-run computes an incorrect result: $x = 2, y = 2, z = 1, u = \mathtt{ff}$. Thus, to ensure idempotent execution, an intermittent system must checkpoint, i.e., save the value of, both volatile and nonvolatile memory. We next explain correct execution of the program in Fig. 1 for atomic and JIT modes.

Atomic Region Execution. As EHDs are highly resource constrained, the system should save state judiciously; checkpointing all of nonvolatile memory is

```
1   Ckpt[a1; x,z:read-only](
2       y:=y+z;
3       let w= x-y in
4           if w>0 then
5               u:=tt
6           else
7               u:=ff);
8       let w=not u in
9           if w then
10              x=x+y;
11              w=ff
12          else
13              skip;
14      skip
```

Fig. 1. An example program with an atomic region and a JIT region



(0) NV_0
ℓ_1 ℓ_2 ℓ_3 ℓ_4
2 0 1 ff
$\gamma := x \mapsto \ell_1, y \mapsto \ell_2, z \mapsto \ell_3, u \mapsto \ell_4$
Initial state
$\uparrow C_{Unit}$
$\Omega_0 := x{:}\uparrow i@CK, y{:}\uparrow i@CK, z{:}\uparrow i@CK, u{:}\uparrow b@CK$
InitWorld
(1) NV_1
ℓ_1 ℓ_2^{ck} ℓ_3 ℓ_4^{ck}
V_1
ℓ_2 ℓ_4
L1
C_{Unit}
$\Omega_{1,2} := x{:}\uparrow i@RD, y^{ck}{:}\uparrow i@CK, z{:}\uparrow i@RD, u^{ck}{:}\uparrow b@CK$
(2) NV_2
V_2
L2
C_{Unit}
$\Sigma_{1,2} := y{:}\downarrow\uparrow i@CK, u{:}\downarrow\uparrow b@CK$
Crash
(3) NV_c
$nat \rightsquigarrow\uparrow C_{Unit}$
$\Omega_c := x{:}\uparrow i@RD, y^{ck}{:}\uparrow i@CK, z{:}\uparrow i@RD, u^{ck}{:}\uparrow b@CK$
Restore
(4) NV_3
V_3
L1-L6
C_{Unit}
$\Omega_3 := x{:}\uparrow i@RD, y^{ck}{:}\uparrow i@CK, z{:}\uparrow i@RD, u^{ck}{:}\uparrow b@CK$
$\Sigma_3 := y{:}\downarrow\uparrow i@RD, u{:}\downarrow\uparrow b@CK$
(5) NV′
V′
ℓ_5
L7
C_{Unit}
$\Omega' := x{:}\uparrow i@RD, y^{ck}{:}\uparrow i@CK, z{:}\uparrow i@RD, u^{ck}{:}\uparrow b@CK$
$\Sigma' := y{:}\downarrow\uparrow i@CK, u{:}\downarrow\uparrow b@CK, w{:}\downarrow\uparrow i@CK$
FinWorld
(6) NV_f
2 1 1 tt
$\gamma := \cdots, w \mapsto \ell_5$
Final state
$\uparrow Unit$
$\Omega_f := x{:}\uparrow i@CK, y{:}\uparrow i@CK, z{:}\uparrow i@CK, u{:}\uparrow b@CK$

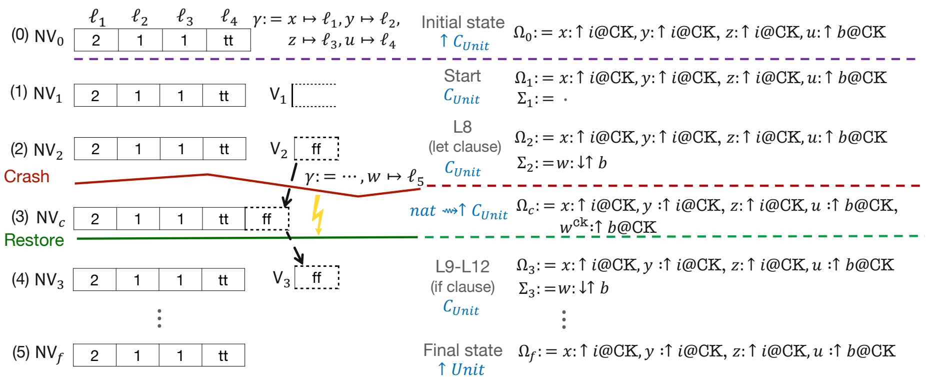
Fig. 2. Intermittent execution of an atomic region. We write i for `int` and b for `bool`.

expensive and unnecessary. For example, variables in an atomic region that are read-only (i.e., never updated) do not change value and need not be checkpointed. In our example, x and z are read-only, so checkpointing y and u is enough to ensure correct intermittent execution. Many intermittent systems follow this design of checkpointing all variables that are not read-only [37,19,17,26,44,12]. Given such a system, Fig. 2 shows an execution of the atomic region in Fig. 1. For now, ignore the last two columns about typing. To save and restore state, the system follows redo-log semantics. It records updates to checkpointed variables in a special volatile region, not main memory. This region clears if power fails, throwing out partial updates. Upon reaching the next atomic or JIT region, the system commits the updates by copying them back to main memory.

Row (0) shows initial nonvolatile locations, their values, and the mapping between variables and memory locations; locations ℓ_1, ℓ_2, ℓ_3, and ℓ_4 in the nonvolatile memory correspond to variables x, y, z and u, respectively. When starting the atomic region (Row (1)), the system takes a snapshot of ℓ_2 and ℓ_4 and stores it in the volatile region V_1. We mark the original nonvolatile locations as checkpointed with the superscript `ck`. i.e., ℓ_2^{ck} and ℓ_4^{ck}.Checkpointed locations ℓ_2^{ck} and ℓ_4^{ck} remain untouched for the remainder of the atomic region execution. Every access to variables y and u will instead be associated with their volatile copy ℓ_2 and ℓ_4, e.g., the assignment in Line 2 is applied to the volatile logs of Row (2).

On power failure, all volatile memory clears (Row (3)), throwing out the log. The system shuts down until more energy is harvested, at which point the system regenerates the volatile copies ℓ_2 and ℓ_4 (Row (4)) and resumes execution from Line 2. When the execution of the atomic region is complete (Row (5)), the system commits the updated values of the checkpointed locations (ℓ_2 and ℓ_4) from volatile memory to their original nonvolatile locations (Row (6)). During execution, local variables are stored to volatile memory via a `let` construct, e.g., location ℓ_5 for variable w on Line 3, corresponding to a volatile execution stack. On power failure, the device clears all volatile memory, but such stack allocated locations will be recreated upon re-execution.

JIT Region Execution. The JIT execution model prevents re-execution, so the intermittent system only saves and restores volatile state at checkpoints. Fig. 3 shows the details of executing the code on the right of Fig. 1 in JIT mode. Row (0) shows the initial nonvolatile locations, their values, and the mapping from variables to locations. The system starts the JIT region by creating an empty context to be populated by volatile locations (Row (1)). The `let` construct in Line 8 allocates a fresh location ℓ_5 in V_2 and updates the mapping to associate variable w to ℓ_5. On a power failure in JIT mode, the system creates a nonvolatile copy of the volatile location ℓ_5 just before it loses the location (Row (3)). It marks the nonvolatile copy with the superscript `ck`. When restoring the program, the system restores these copies to volatile memory and dismisses the nonvolatile backups (Row (4)). The program then continues with the `if` clause on lines 9-12, finally dropping the volatile location ℓ_5, as it is out of scope (Row (5)).



(0) NV$_0$
ℓ_1 ℓ_2 ℓ_3 ℓ_4
2 1 1 tt
$\gamma := x \mapsto \ell_1, y \mapsto \ell_2, z \mapsto \ell_3, u \mapsto \ell_4$
Initial state
$\uparrow C_{Unit}$
$\Omega_0 := x{:}\uparrow i@\mathrm{CK}, y{:}\uparrow i@\mathrm{CK}, z{:}\uparrow i@\mathrm{CK}, u{:}\uparrow b@\mathrm{CK}$
(1) NV$_1$
V$_1$
Start
C_{Unit}
$\Omega_1 := x{:}\uparrow i@\mathrm{CK}, y{:}\uparrow i@\mathrm{CK}, z{:}\uparrow i@\mathrm{CK}, u{:}\uparrow b@\mathrm{CK}$
$\Sigma_1 := \cdot$
(2) NV$_2$
V$_2$ ff
L8
(let clause)
C_{Unit}
$\Omega_2 := x{:}\uparrow i@\mathrm{CK}, y{:}\uparrow i@\mathrm{CK}, z{:}\uparrow i@\mathrm{CK}, u{:}\uparrow b@\mathrm{CK}$
$\Sigma_2 := w{:}\downarrow\uparrow b$
Crash
$\gamma := \cdots, w \mapsto \ell_5$
(3) NV$_c$
ff
$nat \rightsquigarrow \uparrow C_{Unit}$
$\Omega_c := x{:}\uparrow i@\mathrm{CK}, y{:}\uparrow i@\mathrm{CK}, z{:}\uparrow i@\mathrm{CK}, u{:}\uparrow b@\mathrm{CK}, w^{\mathrm{ck}}{:}\uparrow b@\mathrm{CK}$
Restore
(4) NV$_3$
V$_3$ ff
L9-L12
(if clause)
C_{Unit}
$\Omega_3 := x{:}\uparrow i@\mathrm{CK}, y{:}\uparrow i@\mathrm{CK}, z{:}\uparrow i@\mathrm{CK}, u{:}\uparrow b@\mathrm{CK}$
$\Sigma_3 := w{:}\downarrow\uparrow b$
(5) NV$_f$
Final state
$\uparrow Unit$
$\Omega_f := x{:}\uparrow i@\mathrm{CK}, y{:}\uparrow i@\mathrm{CK}, z{:}\uparrow i@\mathrm{CK}, u{:}\uparrow b@\mathrm{CK}$

Fig. 3. Intermittent execution of a JIT region. We write i for `int` and b for `bool`.

3 Key Ideas of Crash Types

We present the intuition behind the stable and unstable memory types (Sec. 3.1), Crash types which internalize checkpointing, power failure/crash, restoration, re-execution, and finalization of atomic regions (Sec. 3.2), and the independence principle applied to intermittent computing (Sec. 3.3).

3.1 Modal Store Types

An unstable value is an intermediate result of an execution towards a stable value and will be lost upon a power failure. However, if the result of a partial execution is committed to a nonvolatile location, it will persist and is thus stable. To reflect the behavior of a memory location in its type, we introduce two (adjoint) modalities $\uparrow_u^s$ (read as "up shift from unstable to stable") and $\downarrow_u^s$ (read as "down shift from stable to unstable"), where $\uparrow_u^s \tau$ indicates that the location stores a stable value of type τ and $\downarrow_u^s \tau$ indicates that the location stores an intermediate result of an execution toward a value of type τ. To fully capture how intermittent execution interacts with a memory location, we also annotate the type of a memory location with an access qualifier, `RD` or `CK`, that represents whether the location is read-only or checkpointed by the system, respectively.

In our example in Fig. 2, the read-only variable x is stored in nonvolatile memory, so it has type $x : \uparrow_u^s \texttt{int@RD}$. The checkpointed variable y has type $y^{\texttt{ck}} : \uparrow_u^s \texttt{int@CK}$ in the nonvolatile memory, while y's volatile copy has type $y : \downarrow_u^s \uparrow_u^s \texttt{int@CK}$. We use the context Ω to type nonvolatile memory and the context Σ to type volatile memory, as shown in the third columns of Figs. 2 and 3. We drop the superscript s and subscript u from the modalities for brevity.

3.2 Crash Types

To capture the effects of intermittent execution in the type of expressions and commands, we introduce *Crash types*, as the notion of stable and unstable values is insufficient. One might expect the expression $x - y$ to have the type $\downarrow\uparrow\texttt{int}$ as it is a (partial) execution towards computing a stable integer value. However, this type does not account for steps due to power failure: the crash itself, waiting for the device to charge, restoration, and re-execution. To reflect these runtime system steps at the type level, we assign the expression a type in the form of a disjunction $\boxed{?} \vee \downarrow\uparrow\texttt{int}$, where $\boxed{?}$ is a type for computations that handle power failures. This type means that the expression either power fails, or completes its execution that evaluates to `int`. Next, we fill in $\boxed{?}$ for commands and expressions. $\boxed{?}$ is a recursive type since it handles re-execution.

Commands. The Crash type for commands is: $\texttt{C}_{\texttt{unit}} = \downarrow(\texttt{nat} \rightsquigarrow \uparrow \texttt{C}_{\texttt{unit}}) \vee \downarrow\uparrow\texttt{unit}$. The right disjunct states that if no power failure occurs while executing a command, then it computes a stable value of type `unit`. The left disjunct states that on power failure, the computation continues as a function; after receiving a (logical) energy input from the environment, it becomes a computation that yields a stable value of a command type, i.e., $\texttt{C}_{\texttt{unit}}$. This computation will execute after the restore, which differs for atomic and JIT modes. In an atomic region, the system re-executes the region from the beginning, and in a JIT region, the system continues with the same command that was interrupted by the failure.

Expressions. The definition of the Crash type for expressions depends on the execution mode, just as the continuation of the program after a power failure depends on the mode. In an atomic region, the system restores an interrupted run of the expression to the original command enclosed in the region, so the type of an atomic mode expression is $\mathtt{C}_A^{\mathsf{atom}} = \downarrow(\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}}) \vee \downarrow\uparrow A$, where the left disjunct is the same as that of a command. On the other hand, an interrupted run of an expression in JIT mode will be restored to the expression itself. Hence, the type of a JIT mode expression is $\mathtt{C}_A^{\mathsf{jit}} = \downarrow(\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_A^{\mathsf{jit}}) \vee \downarrow\uparrow A$, where the left disjunct states that after power failure and reception of the energy input, the computation again yields a stable value of a JIT mode expression type.

3.3 Independence Principle for Typing Intermittent Execution

We design our typing rules to follow the rules for $\downarrow$ and $\uparrow$ modalities in adjoint logic. We introduce two judgment categories. The first category (J_s) is for deriving stable types and corresponds to the judgments of the form $\Omega \vdash \tau^s$, meaning that the rules can rely only on stable locations to evaluate computation on a stable type. The second category (J_u) is for deriving unstable types and corresponds to the judgments of form $\Omega; \Sigma \vdash \tau^u$, meaning that the rules can rely on both stable and unstable locations to evaluate computation on an unstable type.

The adjoint modalities allow going back and forth between judgments J_s and J_u, mirroring checkpointing and restoration operations. The following four sequent calculus rules in the underlying logic govern this back-and-forth behavior in our system. The rules are derivable from the more general rules in prior work [8,34,36]—in particular, the $\uparrow L^*$ rule can be derived from a cut rule and $\downarrow L$. Typical of sequent calculus style rules, we read them bottom-up and match each execution step of a command with the reading of a corresponding rule. Next, we illustrate this matching using the execution steps in Figs. 2 and 3.

$$\frac{\Omega; \cdot \vdash \tau^u}{\Omega \vdash \uparrow\tau^u}\ \uparrow R \qquad \frac{\Omega, \uparrow A^u; \Sigma, \downarrow\uparrow A^u \vdash \tau^u}{\Omega, \uparrow A^u; \Sigma \vdash \tau^u}\ \uparrow L^* \qquad \frac{\Omega \vdash \tau^s}{\Omega; \Sigma \vdash \downarrow\tau^s}\ \downarrow R \quad \frac{\Omega, \uparrow A^u; \Sigma \vdash \tau^u}{\Omega; \Sigma, \downarrow\uparrow A^u \vdash \tau^u}\ \downarrow L$$

Shifts in Atomic Mode (Fig. 2): A combination of $\uparrow R$ and two $\uparrow L^*$ rules corresponds to creating a volatile log from the nonvolatile locations when starting the atomic region, i.e., the step from Row (0) to Row (1). The last two columns in Row (0) correspond to the conclusion of a $\uparrow R$ rule: $\Omega_0 \vdash \uparrow \mathtt{C}_{\mathtt{unit}}$. An application of $\uparrow R$ from bottom to top drops the $\uparrow$ modality from the type of the program and opens an empty volatile region, i.e., $\Omega_0; \cdot \vdash \mathtt{C}_{\mathtt{unit}}$. Next, one application of $\uparrow L^*$, copies the variable y of type $\uparrow \mathtt{int}$ to the volatile memory with the type $\downarrow\uparrow \mathtt{int}$. Similarly, the next application of $\uparrow L^*$ copies the variable u of type $\uparrow$ `bool` to the volatile memory with the type $\downarrow\uparrow$`bool`. The same combination corresponds to creating a volatile log from a nonvolatile location when restarting the atomic region, i.e., the step from Row (3) to Row (4), again copying variables y and u to the volatile memory.

The $\downarrow R$ rule corresponds to a power failure, which erases the volatile memory Σ. From Row (2) to Row (3) in Fig. 2, the system loses the volatile locations of y

and u and closes off the volatile context. Row (2) corresponds to the conclusion of the rule, and Row (3) corresponds to its premise. The type of the command in Row (2) changes from $\texttt{C}_{\texttt{unit}}$ to $\downarrow(\texttt{nat} \rightsquigarrow \uparrow\texttt{C}_{\texttt{unit}})$ (by another $\vee$-R rule as a crash is detected), and then to the type $(\texttt{nat} \rightsquigarrow \uparrow\texttt{C}_{\texttt{unit}})$ in Row (3).

Finally, a $\downarrow L$ rule combined with a standard weakening rule and a $\downarrow R$ rule corresponds to the final commit of the volatile context, i.e., stepping from Row (5) to Row (6), the nonvolatile context drops the locations y and u of types $\uparrow\texttt{int}$ and $\uparrow\texttt{bool}$, respectively, by a weakening rule. These two variables map to the locations with outdated values. Next, the volatile locations of y and u in Σ', which contain the up-to-date values, commit their values to the nonvolatile context by a $\downarrow L$ rule. Then, a $\downarrow R$ rule closes off the remaining volatile context, which contains w of type $\downarrow\uparrow\texttt{int}$. The type of the command in Row (2) changes from $\texttt{C}_{\texttt{unit}}$ to $\downarrow\uparrow\texttt{unit}$ (by a separate $\vee$-R rule as the system detects a successful execution) and from that to type $\uparrow\texttt{int}$ in Row (6).

Shifts in JIT Mode (Fig. 3): A $\uparrow R$ rule corresponds to creating an empty volatile context Σ_1 when starting the JIT region, i.e., the step from Row (0) to Row (1). A combination of the $\downarrow L$ rule and $\downarrow R$ rule corresponds to a power failure, i.e., the stepping from Row (2) to Row (3). A $\downarrow L$ rule copies the location w of type $\downarrow\uparrow\texttt{bool}$ from volatile memory Σ_2 to nonvolatile memory Ω_c. A $\downarrow R$ rule closes off the (empty) nonvolatile memory. As in atomic mode, a combination of $\uparrow R$ and $\uparrow L^*$ rules corresponds to creating a volatile log from a nonvolatile location when restarting the command after the failure, i.e., the step from Row (3) to Row (4). The $\uparrow R$ rule clears a portion of volatile memory, and the $\uparrow L^*$ rule copies variable w from nonvolatile memory into volatile memory. We need an extra weakening rule to eliminate the remaining variable w in nonvolatile memory. The dropping of volatile memory at the end of execution (Row (5)) is not a modal step, but rather follows from a standard rule for the let clause.

4 A Basic Calculus for Intermittent Execution

We present the syntax, semantics, and the Crash type system for a basic calculus.

4.1 Syntax

The syntactic constructs are summarized in Fig. 4. Expressions include constants, variables, and binary operations while commands include assignments, mutable let bindings, sequencing, and if branching. A program consists of sequenced blocks of commands and atomic regions, denoted $\mathsf{Ckpt}[\mathsf{aID}, \rho](c)$ with a unique identifier aID, read-only variables ρ, and the enclosed command c.

Nonvolatile memory (NV) and volatile memory (V) map locations ℓ to values. Each location is annotated with its access mode q ($\texttt{RD}$ or $\texttt{CK}$). The nonvolatile memory location $\ell_{\texttt{ck}}$ is the checkpointed copy of location ℓ in volatile memory. The context γ maps variable names to memory locations. Access mode qualifiers in V and NV have constrained values (to be discussed in the semantics).

Command, expression, and memory

values	v	$::= \mathsf{n} \mid \mathsf{tt} \mid \mathsf{ff} \mid x$	*access qualifier*	q	$::= \mathtt{CK} \mid \mathtt{RD}$
exprs	e	$::= v \mid e \odot e$	*var loc map*	γ	$::= \cdot \mid \gamma, x \mapsto \ell$
cmds	c	$::= \mathsf{skip} \mid \mathsf{let}\, x = e \,\mathsf{in}\, c \mid c; c$	*nonvolatile mem*	NV	$::= \cdot \mid \ell \,@\, q \hookrightarrow v, \mathsf{NV}$
		$\mid\ \mathsf{if}\, e \,\mathsf{then}\, c \,\mathsf{else}\, c \mid x ::= e$			$\mid\ \ell_{\mathsf{ck}} \,@\, \mathtt{CK} \hookrightarrow v, \mathsf{NV}$
progs	p	$::= \mathsf{Ckpt}[\mathsf{aID}, \rho](c); p \mid c; p \mid \mathsf{skip}$	*volatile mem*	V	$::= \cdot \mid l \,@\, \mathtt{CK} \hookrightarrow v, \mathsf{V}$

Instructions, statements, and configurations.

commands	c	$::= \cdots c;_W c$	*crash instrs*	i	$::= \downharpoonleft\varepsilon \;\texttt{\#}\; \mathsf{in}(b > 0, \upharpoonleft\kappa)$
continuations	κ	$::= c \mid e$			$\mid\ \varepsilon \;\texttt{\#}\; \mathsf{in}(b > 0, \upharpoonleft\kappa) \mid \upharpoonleft \kappa$
statements	s	$::= \kappa \mid i \mid p$	*open config*	K_o	$::= (\gamma \mid \mathtt{Md} \mid g \mid \mathsf{NV} \mid \mathsf{V} \mid s)$
energy level	g	$::= \cdot \mid n$			$\mid\ (\gamma \mid \mathtt{Md} \mid g \mid \mathsf{NV} \mid s)$
charge stream	χ	$::= n :: \chi$	*closed config*	K_c	$::= [\chi \rhd \varepsilon] \otimes K_o$
exec. mode	$\mathtt{Md}$	$::= \mathsf{aID}(c) \mid \mathsf{jit}$			

Fig. 4. Summary of syntax

The runtime instruction $c_1;_W c_2$ is used for evaluating c_1 under the execution context W. To model energy harvesting from the environment, we assume a unique external energy channel, ε, from which the system receives energy. Three crash instructions control the system in the event of a power failure. The instruction $\downharpoonleft\varepsilon \;\texttt{\#}\; \mathsf{in}(b > 0, \upharpoonleft\kappa)$ models the system that faces a power failure, where κ is the interrupted command or expression, and $b > 0$ is a guard to ensure that the bound incoming energy variable b is positive. The instruction $\varepsilon \;\texttt{\#}\; \mathsf{in}(b > 0, \upharpoonleft\kappa)$ models the system awaiting an energy input to be bound to b. The instruction $\upharpoonleft\kappa$ models the system ready to restore memory and re-execute.

We write K_o to denote an *open* system configuration, consisting of the mapping γ, the mode of execution $\mathtt{Md}$ (i.e., atomic or JIT), energy available for this execution g, memories, and the statement s to be executed. The energy level $(\cdot)$ models the state right after power failure. We close an open configuration with $[\chi \rhd \varepsilon]$; we connect it via an external energy channel ε to an infinite charging stream Ξ of natural numbers, which models available energy the configuration harvests from the environment at each power failure point for re-execution.

We call a configuration that cannot take a step a value configuration (value for short). An open configuration of form $(\cdots \mid g \mid \cdots \mid s)$ is a value, i.e., $\mathit{Val}(\cdots \mid g \mid \cdots \mid s)$, if either s is a constant or skip, it has depleted all energy for this execution $(g = 0)$, or s is a crash instruction. The latter two cases are values because they cannot take a step without interacting with the environment or perform operations on the volatile and novolatile memory specific to handling power failures. A closed configuration is a value only if the statement s is skip with some energy left $(g > 0)$. We list all values in the extended TR [15].

4.2 Operational Semantics

Top-level Program Execution. The top-level semantic rules for setting up and finalizing the atomic and JIT execution contexts are shown in Fig. 5. The P-CKPT rule applies if the next code block is an atomic region. The nonvolatile

$$\frac{\begin{array}{c} n > 0 \quad \mathsf{InitWorld}_d(\mathsf{NV};\rho;\gamma) = \mathsf{NV}_0, V_0 \\ [\chi \rhd \varepsilon] \otimes \gamma \,|\, \mathsf{aID}(c_0) \,|\, n \,|\, \mathsf{NV}_0 \,|\mathsf{V}_0|\, c_0 \Rightarrow^* [\chi' \rhd \varepsilon] \otimes \gamma' \,|\, \mathsf{aID}(c_0) \,|\, n' |\, \mathsf{NV}' |\, \mathsf{V}' |\, \mathsf{skip} \\ n' > 0 \quad \mathsf{NV}_1 = \mathsf{FinWorld}_d(\mathsf{NV}';\mathsf{V}') \end{array}}{[\chi \rhd \varepsilon] \otimes \gamma \mid n \mid \mathsf{NV} \mid \mathsf{Ckpt}[(\mathsf{aID};\rho)](c_0); p \;\Rightarrow\; [\chi' \rhd \varepsilon] \otimes \gamma \mid n' \mid \mathsf{NV}_1 \mid p} \text{(P-CKPT)}$$

$$\frac{\begin{array}{c} n > 0 \quad n' > 0 \\ [\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{jit} \mid n \mid \mathsf{NV} \mid \cdot \mid c \Rightarrow^* [\chi' \rhd \varepsilon] \otimes \gamma' \mid \mathsf{jit} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid \mathsf{skip} \end{array}}{[\chi \rhd \varepsilon] \otimes \gamma \mid n \mid \mathsf{NV} \mid c; p \;\Rightarrow\; [\chi' \rhd \varepsilon] \otimes \gamma \mid n' \mid \mathsf{NV}' \mid p} \text{(P-SEQ)}$$

Fig. 5. Closed configuration semantics for programs

NV_0 and volatile V_0 locations are initialized based on a given NV, declared read-only variables ρ, and their mapping γ to locations. The $\mathsf{InitWorld}_d$ function (a) changes the qualifier of locations in NV that are declared as read-only in ρ from `CK` to `RD`, (b) creates V_0 by copying the rest of the locations of NV that still have qualifier `CK`, and (c) marks the original version of the locations ℓ in NV that still have qualifier `CK` as checkpointed ($\ell_{\mathtt{ck}}$). This part corresponds to the step from Row (0) to Row (1) in Fig. 2. The closed configuration of c_0 is evaluated until completion, using the rules in Fig. 6. This execution may undergo several power failures and corresponds to the steps from Row (1) to Row (5) in Fig. 2. Finally, the $\mathsf{FinWorld}_d$ function closes off atomic regions, finalizing the volatile and nonvolatile locations. $\mathsf{FinWorld}_d$ (a) copies the values of volatile locations in V' that have a checkpointed version into NV', (b) removes `CK` from the locations in NV', i.e., converts $\ell_{\mathtt{ck}}$ to ℓ, and (c) replaces the `RD` qualifier of the locations in NV' with `CK`. This corresponds to the step from Row (5) to Row (6) in Fig. 2.

The P-SEQ rule applies when the next code block is a regular command c. The closed configuration of c with an empty initial set of volatile locations is fully evaluated. This corresponds to the steps from Row (0) to Row (1) and Row (1) to Row (5) in Fig. 3. Then the resulting volatile locations V' scoped in c are dropped, corresponding to the step from Row (5) to Row (6) in Fig. 3.

Command Execution (Closed Config). We summarize rules for a closed configuration in the top part of Fig. 6. Rule D-STEP steps the closed command configuration when the corresponding open configuration steps. Next, we explain the trio of power failure, charge, and restore rules. When the energy for this execution is depleted (i.e., $g = 0$), the D-CRASH rule applies, stepping the system to the crash instruction $\downarrow\varepsilon$ # $\mathsf{in}(b > 0; \uparrow\kappa)$. Next, D-S-JIT or D-S-AID rules apply and operate on volatile memory based on the execution mode `Md`. In JIT mode, D-S-JIT checkpoints and stores all volatile memory in nonvolatile locations. In atomic mode, D-S-AID drops all volatile memory locations. Then, D-CHARGE applies and inputs a natural number $n > 0$ from the energy channel, replenishing the configuration's energy level for re-execution. Finally, the program is restored via D-RESTORE-JIT and D-RESTORE-AID which copy checkpointed locations into volatile memory. D-RESTORE-JIT drops the checkpointed regions and steps

Closed Configuration Semantics for Commands and Crash Instructions

$$\frac{\gamma \,|\, \texttt{Md} \,|\, n \,|\, \mathsf{NV} \,|\, \mathsf{V} \,|\, c \rightarrow\ \gamma \,|\, \texttt{Md} \,|\, n' \,|\, \mathsf{NV}' \,|\, \mathsf{V}' \,|\, c'}{[\chi \rhd \varepsilon] \otimes \gamma \,|\, \texttt{Md} \,|\, n \,|\, \mathsf{NV} \,|\, \mathsf{V} \,|\, c \Rightarrow\ [\chi \rhd \varepsilon] \otimes \gamma \,|\, \texttt{Md} \,|\, n' \,|\, \mathsf{NV}' \,|\, \mathsf{V}' \,|\, c'}\ \text{(D-STEP)}$$

$$\frac{}{[\chi \rhd \varepsilon] \otimes \gamma \,|\, \texttt{Md} \,|\, 0 \,|\, \mathsf{NV} \,|\, \mathsf{V} \,|\, c \Rightarrow [\chi \rhd \varepsilon] \otimes \gamma \,|\, \texttt{Md} \,|\, \cdot \,|\, \mathsf{NV} \,|\, \mathsf{V} \,|\, \downarrow \varepsilon \;\#\; \mathsf{in}(b > 0; \uparrow c)}\ \text{(D-CRASH)}$$

$$\frac{\texttt{Md} = \mathsf{jit}}{\begin{array}{l}[\chi \rhd \varepsilon] \otimes \gamma \,|\, \texttt{Md} \,|\, \cdot \,|\, \mathsf{NV} \,|\, \mathsf{V} \,|\, \downarrow\varepsilon \;\#\; \mathsf{in}(b > 0; \uparrow\kappa) \\ \Rightarrow\ [\chi \rhd \varepsilon] \otimes \gamma \,|\, \texttt{Md} \,|\, \mathsf{NV}, \mathsf{V}_{\mathsf{ck}} \,|\, \varepsilon \;\#\; \mathsf{in}(b > 0; \uparrow\kappa)\end{array}}\ \text{(D-S-JIT)}$$

$$\frac{\texttt{Md} = \mathsf{aID}(c_0) \quad \gamma' \subseteq \gamma \quad \mathit{range}(\gamma') = \mathit{dom}(\mathsf{NV})}{\begin{array}{l}[\chi \rhd \varepsilon] \otimes \gamma \,|\, \texttt{Md} \,|\, \cdot \,|\, \mathsf{NV} \,|\, \mathsf{V} \,|\, \downarrow\varepsilon \;\#\; \mathsf{in}(b > 0; \uparrow\kappa) \\ \Rightarrow\ [\chi \rhd \varepsilon] \otimes \gamma' \,|\, \texttt{Md} \,|\, \cdot \,|\, \mathsf{NV} \,|\, \varepsilon \;\#\; \mathsf{in}(b > 0; \uparrow\kappa)\end{array}}\ \text{(D-S-AID)}$$

$$\frac{}{[n :: \chi \rhd \varepsilon] \otimes \gamma \,|\, \texttt{Md} \,|\, \cdot \,|\, \mathsf{NV} \,|\, \varepsilon \;\#\; \mathsf{in}(b > 0; \uparrow\kappa) \Rightarrow [\chi \rhd \varepsilon] \otimes \gamma \,|\, \texttt{Md} \,|\, n \,|\, \mathsf{NV} \,|\, \uparrow\kappa}\ \text{(D-CHARGE)}$$

$$\frac{\mathsf{NV} = \mathsf{NV}', \mathsf{NV}''_{\mathsf{ck}}}{[\chi \rhd \varepsilon] \otimes \gamma \,|\, \mathsf{jit} \,|\, n \,|\, \mathsf{NV} \,|\, \uparrow \kappa\ \Rightarrow\ [\chi \rhd \varepsilon] \otimes \gamma \,|\, \mathsf{jit} \,|\, n \,|\, \mathsf{NV}' \,|\, \mathsf{NV}'' \,|\, \kappa}\ \text{(D-RESTORE-JIT)}$$

$$\frac{\mathsf{NV} = \mathsf{NV}', \mathsf{NV}''_{\mathsf{ck}}}{[\chi \rhd \varepsilon] \otimes \gamma \,|\, \mathsf{aID}(c_0) \,|\, n \,|\, \mathsf{NV} \,|\, \uparrow \kappa \Rightarrow [\chi \rhd \varepsilon] \otimes \gamma \,|\, \mathsf{aID}(c_0) \,|\, n \,|\, \mathsf{NV} \,|\, \mathsf{NV}'' \,|\, c_0}\ \text{(D-RESTORE-AID)}$$

Selected expression and command semantics

$$\frac{\gamma = \gamma', [x \mapsto \ell] \quad \mathsf{V} = \ell@q \hookrightarrow v, \mathsf{V}' \quad n = n' + 1}{\gamma \mid \texttt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid x \rightarrow \gamma \mid \texttt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid v}\ \text{(D-V-READ)}$$

$$\frac{\begin{array}{c}\mathit{Val}(\gamma \mid \texttt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e) \\ \mathsf{V} = \mathsf{V}', \ell@q \hookrightarrow v' \quad q \neq \texttt{RD} \quad \gamma = \gamma', [x \rightarrow \ell] \quad n = n' + 1\end{array}}{\gamma \mid \texttt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid x := e\ \rightarrow \gamma \mid \texttt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V}', \ell@q \hookrightarrow e \mid \mathsf{skip}}\ \text{(D-ASSIGN-V)}$$

Fig. 6. Statement steps

to the interrupted command κ, while D-RESTORE-AID keeps the checkpointed regions and steps to the original command c_0 in the atomic region.

Command/Expression Execution (Open Config). The rules for executing commands and expressions in an open configuration are standard. We present a selection of them on the bottom of Fig. 6. Each step decrements the energy level by one. The rules ensure that checkpointed location ℓ_{ck} in NV is not read by the program, as it could store outdated data, and is not written to, as this would tamper with the checkpointed value.

4.3 Types, Typing Contexts, and Judgments

This section introduces the typing judgments used in our static typing.

(J_u)	$\mathtt{Md} \mid b\,\mathcal{R}\,0 : \mathtt{nat} \mid \Omega; \Sigma \vdash c :: \mathtt{C}_{\mathtt{unit}}$	c could crash
(J_u)	$\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega; \Sigma \vdash \mathsf{skip} :: \downarrow\uparrow\mathtt{unit}$	c will not crash
(J_s)	$\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash \mathsf{skip} :: \uparrow\mathtt{unit}$	after commit
(J_u)	$\mathtt{Md} \mid b\,\mathcal{R}\,0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD}} e :: \mathtt{C}_A^{\mathtt{Md}}$	e read, could crash
(J_s)	$\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD}} v :: \downarrow\uparrow A$	e read no crash
(J_s)	$\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathtt{RD}} v :: \uparrow A$	e read, commit
(J_u)	$\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{WT}} x :: \downarrow\uparrow A$	write on x, no crash
(J_s)	$\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathtt{WT}} x :: \uparrow A$	write on x, commit
(J_s)	$\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash p :: \uparrow\mathtt{C}_{\mathtt{unit}}$	before execution
(J_u)	$\mathtt{Md} \mid b = 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash \kappa :: \mathtt{C}_T^{\mathtt{Md}}$	about to crash
(J_u)	$\mathtt{Md} \mid \cdot \mid \Omega; \Sigma \vdash \downarrow\varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow\kappa) :: \downarrow(\mathtt{nat} \rightsquigarrow \uparrow\mathtt{C}_T^{\mathtt{Md}})$	crash state
(J_s)	$\mathtt{Md} \mid \cdot \mid \Omega \vdash \varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow\kappa) :: \mathtt{nat} \rightsquigarrow \uparrow\mathtt{C}_T^{\mathtt{Md}}$	waiting for energy
(J_s)	$\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega \vdash \uparrow\kappa :: \uparrow\mathtt{C}_T^{\mathtt{Md}}$	before re-execution

Table 1. Typing judgment summary

Types and Static Context. Our types are summarized below. The two modalities stratify types into the varieties stable (τ^s) and unstable (τ^u). The base store types `int` and `bool` are considered unstable. A type variable v_t denotes a type in the set $\{\mathtt{C}_{\mathtt{unit}}, \mathtt{C}_A^{\mathsf{atom}}, \mathtt{C}_A^{\mathsf{jit}}\}$, and implements the recursive nature of Crash types. We include the connectives $\vee$ and $\rightsquigarrow$ solely for the purpose of defining Crash types; they are not used elsewhere. Defining Crash types using these connectives will allow us to define the logical relation in Sec. 5 based on the intended meaning of its index type. Some well-formed types, e.g., $\mathtt{nat} \rightsquigarrow \mathtt{nat} \rightsquigarrow \uparrow\mathtt{unit}$, are not accepted by our type system introduced in Sec. 4.4. These types have no inhabitants, i.e., no well-typed configuration is of these types.

$$\begin{array}{llll}
\textit{store types } A := \mathtt{int} \mid \mathtt{bool} & \textit{stable types} & \tau^s & := \mathtt{nat} \rightsquigarrow \tau^s \mid \uparrow \tau^u \\
\textit{basic types } T := \mathtt{unit} \mid A & \textit{unstable types} & \tau^u & := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t \\
\textit{Volatile store typing context} & & \Sigma & := \cdot \mid x : \downarrow_u^s \uparrow_u^s A@Ck,\, \Sigma \\
\textit{Nonvolatile store typing context} & & \Omega & := \cdot \mid x : \uparrow_u^s A@Rd,\, \Omega \mid x_{\mathtt{ck}} : \uparrow_u^s A@\mathtt{CK}, \Omega \\
& & & \quad \mid x : \uparrow_u^s A@\mathtt{CK}, \Omega
\end{array}$$

A nonvolatile store typing context Ω assigns stable types to nonvolatile location variables, i.e. all variables in Ω have a type of the form $\uparrow_u^s A$. A volatile store typing context Σ assigns unstable types to volatile location variables, i.e., variables in Σ are of the type $\downarrow_u^s \uparrow_u^s A$. $x_{\mathtt{ck}}$ refers to a location that has been checkpointed. In the atomic mode, $x_{\mathtt{ck}}$ has an active volatile log in Σ.

Typing Judgments. Table 1 summarizes all the typing judgments. These judgments are parameterized over the execution mode `Md` of the expression or command to be typed. The judgment also tracks a variable b corresponding to the current energy level of this execution. b ranges over natural numbers (`nat`) and is constrained by a relation $\mathcal{R} \in \{\geq, >\}$ or is set to 0; where $b \geq 0$ is unconstrained. The constraint on b determines whether or not a command can evaluate a value without power failure. There are three judgments for command typing. The first judgment is used when the command has not yet successfully finished

$$\frac{\mathsf{jit} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \cdot \vdash_{\emptyset} c : \mathsf{C}_{\mathtt{unit}} \quad b : \mathtt{nat} \mid \Omega \vdash p : \uparrow\mathsf{C}_{\mathtt{unit}}}{b : \mathtt{nat} \mid \Omega \vdash c; p : \uparrow\mathsf{C}_{\mathtt{unit}}} \text{(T-P-SEQ)}$$

$$\frac{\begin{array}{c}\Omega_0 \mid \Sigma_0 = \mathsf{InitWorld}_t(\Omega; \rho) \\ \mathtt{Sig} = \{\mathsf{aID}(c_0) \mid b \geq 0 : \mathtt{nat} \mid \Omega_0; \Sigma_0 \vdash c_0 : \mathsf{C}_{\mathtt{unit}}\} \\ \mathsf{aID}(c_0) \mid b \geq 0 : \mathtt{nat} \mid \Omega_0;\ \Sigma_0 \vdash_{\mathtt{Sig}} c_0 : \mathsf{C}_{\mathtt{unit}} \quad b\ : \mathtt{nat} \mid \Omega \vdash p : \uparrow\mathsf{C}_{\mathtt{unit}}\end{array}}{b : \mathtt{nat} \mid \Omega \vdash \mathsf{Ckpt}[\mathsf{aID}, \rho](c_0); p : \uparrow\mathsf{C}_{\mathtt{unit}}} \text{(T-P-CKPT)}$$

Fig. 7. Program typing

executing; its next step, depending on its constraint $\mathcal{R}$, may or may not crash. When the command reaches type $\downarrow\uparrow\mathtt{unit}$, b no longer needs to be constrained as the execution succeeded without power failure. The second judgment invokes the third judgment to type the configuration after the volatile log is committed: in the typing rule for committing the volatile log, the conclusion is of the form of the second judgment and the premise is of the form of the third. For expression typing, we distinguish expressions on the right of an assignment (being read) from those on the left of an assignment (being written to) via subscripts $\mathtt{RD}$ and $\mathtt{WT}$, respectively. The expressions that are being written to are only of the simple form x. As no execution is required to evaluate x, we consider its judgment crash free, so no constraint is required on b. For program typing, we only have one judgment that refers to the type of the program before the execution of its next block starts. The rest of the judgments type states after a crash. The first judgment uses the constraint $b = 0$, which corresponds to the power failure condition. It invokes the second judgment, which types a state right after crash. The third judgment types the state awaiting energy to continue re-execution, and the final judgment types the state that is ready for restoration and re-execution.

4.4 Typing Rules

Program Typing. Fig. 7 shows the typing rules for programs. The P-SEQ rule types program $c; p$ by first typing c under jit mode, requiring $b \geq 0$, and then typing the rest of the program. The volatile memory context is empty for now, but will be populated when the let commands allocate new volatile locations.

The P-CKPT rule types the command c_0 enclosed in an atomic region under the mode $\mathsf{aID}(c_0)$ and then types the rest of the program p. The first premise sets up the initial typing contexts for nonvolatile and volatile memories, as illustrated in Fig. 2. The partial function $\mathsf{InitWorld}_t$ initializes the volatile memory by creating a log of variables in Ω that are not read-only. Ω can be uniquely split into Ω^c and Ω^r, where Ω^r is the set of all read-only locations in Ω, and Ω^c is the set of all locations that are not read-only. This function is defined below:

$\Omega_0 \mid \Sigma_0 = \mathsf{InitWorld}_t(\Omega; \rho)$ iff $\rho \subseteq \mathsf{dom}(\Omega)$, $\Omega_0 = \Omega^r, \Omega^c_{\mathsf{ck}}$ and $\Sigma_0 = \downarrow\Omega^c$
where $\Omega = \Omega^c, \Omega^r$ and $\Omega^r = \Omega\lceil\rho$.

Here $\Omega^r = \Omega\lceil\rho$ is a subset of Ω where locations are declared in ρ to be read-only, and Ω^c are all other locations in Ω. The context Ω^c_{ck}, is defined as

$\Omega^c_{\texttt{ck}} = \{x_{\texttt{ck}} : \uparrow A@q \mid x : \uparrow A@q \in \Omega^c\}$, and the context $\downarrow\Omega^c$, is defined as $\downarrow\Omega^c = \{x : \downarrow\uparrow A@q \mid x : \uparrow A@q \in \Omega^c\}$. If the set of read only variables, ρ, is not in the domain of Ω, then the function $\mathsf{InitWorld}_t$ is not defined.

In rules P-SEQ and P-CKPT, the command typing judgment in the premise makes use of a signature (subscripts $\emptyset$ and $\texttt{Sig}$, respectively) to type check the command relative to the signature. The signature is populated at different stages of type checking the JIT and atomic regions. In an atomic region, rule T-P-CKPT populates the signature at the beginning of the region with the initial judgment which includes the region's original command c_0 and static memory context $\Omega_0; \Sigma_0$. The region is then typed relative to the signature. In JIT mode, the signature is populated later with the judgment just at the point of the failure (rule T-ENOUGH?). The program remembers that it built a typing derivation for the judgment in the signature such that when it restores from a power failure, it refers to the signature and checks that the restored judgment matches the one stored in the signature without needing to derive it again. This makes the typing derivations finitary and inductive.

Command and Expression Typing. Fig. 8 shows selected typing rules for commands. The T-SKIP rule declares the command skip as the stable type $\uparrow\texttt{unit}$. Rule T-$\vee$-SUCC applies when the command successfully completes its execution and still has one unit of energy available ($b > 0$) to conclude the execution. In this case, we close off the energy level variable and continue typing the command against the type $\downarrow\uparrow$ $\texttt{unit}$. Rule T-C-SHIFT is invoked by T-$\vee$-SUCC and updates the memory typing contexts by removing checkpointed locations in Ω as now they are not needed, and making locations in Σ stable as now they are committed. This corresponds to the last step of Fig. 2.

The rules T-LET and T-ASSIGN, are mostly standard except that we consider crashes. For example, in typing the assign command $x := e$, the first premise of T-ASSIGN considers the type of expression e to be the Crash type $C^{\texttt{Md}}_A$, but in the second premise we require the location x to be of type $\downarrow\uparrow A$, i.e., the location only considers the type corresponding to the case where execution of e can be completed successfully. The reason is that the assignment only occurs if the execution of e is successful. The constraint on the energy levels for premises goes back to $b \geq 0$, as we use one energy unit to deconstruct these commands.

The rule T-ENOUGH? checks two premises based on the value of $b \geq 0$. The third premise, a crash judgment, corresponds to the case where $b = 0$ (typing rules for crash judgments are given later in this section) and the fourth premise corresponds to the case where $b > 0$. The condition $b > 0$ states that there is at least one unit of energy available to decompose one command construct, e.g., via T-LET or T-ASSIGN. This rule populates the signature for JIT commands. The second premise states that the signature remains intact if the mode is atomic, but is populated by $\texttt{Sig}'$ if the mode is JIT. In the JIT mode, after a power failure, the command c is restored to itself, and $\texttt{Sig}'$ remembers that the well-typedness of the command when the energy level is non-negative has been checked already.

Expression typing rules are very similar to those of the commands. Fig. 8 shows a few selected rules. The T-LOC-WRITE and T-LOC-READ rules match

Commands

$$\frac{}{\mathtt{Md} \mid b : \mathbf{nat} \mid \Omega \vdash_{\mathtt{Sig}} \mathsf{skip} : \uparrow\mathtt{unit}} \text{ (T-Skip)}$$

$$\frac{\Sigma = \downarrow\Sigma' \quad \Omega = \Omega', \Omega''_{\mathtt{ck}} \quad \mathtt{Md} \mid b : \mathbf{nat} \mid \Omega', \Sigma' \vdash_{\mathtt{Sig}} \mathsf{skip} : \uparrow\mathtt{unit}}{\mathtt{Md} \mid b : \mathbf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \mathsf{skip} : \downarrow\uparrow\mathtt{unit}} \text{ (T-C-Shift)}$$

$$\frac{\mathtt{Md} \mid b : \mathbf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \mathsf{skip} : \downarrow\uparrow\mathtt{unit}}{\mathtt{Md} \mid b > 0 : \mathbf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \mathsf{skip} : \tau \vee \downarrow\uparrow\mathtt{unit}} \text{ (T-}\vee\text{-Succ)}$$

$$\frac{\begin{array}{c}\mathtt{Md} \mid b \geq 0 : \mathbf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e_1 : \mathtt{C}^{\mathtt{Md}}_A \\ \mathtt{Md} \mid b \geq 0 : \mathbf{nat} \mid \Omega; \Sigma, x{:}\downarrow\uparrow A@\mathtt{CK} \vdash_{\mathtt{Sig}} c : \tau\end{array}}{\mathtt{Md} \mid b > 0 : \mathbf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \mathsf{let}\, x = e_1 \,\mathsf{in}\, c : \tau} \text{ (T-Let)}$$

$$\frac{\mathtt{Md} \mid b \geq 0 : \mathbf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e : \mathtt{C}^{\mathtt{Md}}_A \quad \mathtt{Md} \mid b > 0 : \mathbf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{WT}} x : \downarrow\uparrow A}{\mathtt{Md} \mid b > 0 : \mathbf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} x := e : \mathtt{C}^{\mathtt{Md}}_{\mathtt{unit}}} \text{ (T-Assign)}$$

$$\frac{\begin{array}{c}\mathtt{Sig}' = \{\mathtt{Md} \mid b \geq 0 : \mathbf{nat} \mid \Omega; \Sigma \vdash c : \tau\} \\ \mathtt{Sig}'' = \mathit{if}\, \mathtt{Md} = \mathsf{jit}, \mathit{then}\, \mathtt{Sig}', \mathit{else}\, \mathtt{Sig} \\ \mathtt{Md} \mid b = 0 : \mathbf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}''} c : \tau \quad \mathtt{Md} \mid b > 0 : \mathbf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \tau\end{array}}{\mathtt{Md} \mid b \geq 0 : \mathbf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \tau} \text{ (T-enough?)}$$

Expressions

$$\frac{\Omega, \Sigma' = x{:}\uparrow A@q, \Omega'_2 \quad q \neq \mathtt{RD}}{\mathtt{Md} \mid b : \mathbf{nat} \mid \Omega, \Sigma' \vdash_{Wt} x : \uparrow A} \text{ (T-Loc-Write)}$$

$$\frac{\Omega = x : \uparrow A@q, \Omega'}{\mathtt{Md} \mid b : \mathbf{nat} \mid \Omega \vdash_{\mathtt{RD}} x : \uparrow A} \text{ (T-Loc-Read)} \qquad \frac{}{\mathtt{Md} \mid b : \mathbf{nat} \mid \Omega \vdash_{\mathtt{RD}} \mathsf{tt} :\uparrow \mathtt{bool}} \text{ (T-Bool-t)}$$

Fig. 8. Selected command and expression typing

the location variable x with an existing variable inside the context. T-Loc-Write performs an extra check to make sure that x is not a read-only variable.

Statement typing Fig. 9 presents the typing rules for crash instructions. The crash is detected by the depleted energy level $b = 0$ in the T-$\vee$-crash rule. In the premise, the crash instruction $\downarrow\varepsilon$ # $\mathsf{in}(b > 0, \uparrow\kappa')$ is typed. In JIT mode, the T-Jit-stop rule brings a checkpointed version of all the volatile variables in Σ inside Ω since they are checkpointed then. In atomic mode, T-aID-Stop rule simply drops the volatile locations in Σ. The T-charge rule inputs a new energy level from the energy channel ε, regardless of the mode. The first premise shows that the energy channel is needed to provide a natural number greater than zero. Finally, the T-Jit-Restore and T-aID-Restore rules prepare and check rebooted system in JIT and atomic modes, respectively. In both modes, volatile memory is restored from the checkpointed locations in Ω. In the atomic mode, the checkpointed locations persist in Ω as we may need them for the

$$\frac{\mathtt{Md} \mid \cdot \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \downarrow\varepsilon \;\#\; \textsf{iņ}(b > 0, \uparrow\kappa') : \downarrow(\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}^{\mathtt{Md}}_{T'})}{\mathtt{Md} \mid b = 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \kappa' : \downarrow(\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}^{\mathtt{Md}}_{T'}) \vee \downarrow\uparrow T} \text{ (T-}\vee\text{-Crash)}$$

$$\frac{\Sigma = \downarrow\uparrow\Sigma' \quad \textsf{jit} \mid \cdot \mid \Omega, \uparrow\Sigma'_{\mathtt{ck}} \vdash_{\mathtt{Sig}} \varepsilon \;\#\; \textsf{iņ}(b > 0, \uparrow\kappa') : (\mathtt{nat} \rightsquigarrow \uparrow\mathtt{C}^{s}_{T})}{\textsf{jit} \mid \cdot \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \downarrow\varepsilon \;\#\; \textsf{iņ}(b > 0, \uparrow\kappa') : \downarrow(\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}^{s}_{T})} \text{ (T-Jit-stop)}$$

$$\frac{\textsf{aID}(c_0) \mid \cdot \mid \Omega \vdash_{\mathtt{Sig}} \varepsilon \;\#\; \textsf{iņ}(b > 0, \uparrow\kappa') : (\mathtt{nat} \rightsquigarrow \uparrow\mathtt{C}^{s}_{\mathtt{unit}})}{\textsf{aID}(c_0) \mid \cdot \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \downarrow\varepsilon \;\#\; \textsf{iņ}(b > 0, \uparrow\kappa') : \downarrow(\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}^{s}_{\mathtt{unit}})} \text{ (T-aID-stop)}$$

$$\frac{\varepsilon \;\#\; \textsf{iņ}() : \mathtt{nat} > 0 \quad \mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega \vdash_{\mathtt{Sig}} \uparrow \kappa' : \uparrow\mathtt{C}^{s}_{T}}{\mathtt{Md} \mid \cdot \mid \Omega \vdash_{\mathtt{Sig}} \varepsilon \;\#\; \textsf{iņ}(b > 0, \uparrow \kappa') : (\mathtt{nat} \rightsquigarrow \uparrow\mathtt{C}^{s}_{T}))} \text{ (T-Charge)}$$

$$\frac{\Omega = \Omega', \Omega''_{\mathtt{ck}} \quad \textsf{jit} \mid b \geq 0 : \mathtt{nat} \mid \Omega'; \downarrow\Omega'' \vdash \kappa' : \mathtt{C}_T \in \mathtt{Sig}}{\textsf{jit} \mid b > 0 : \mathtt{nat} \mid \Omega \vdash_{\mathtt{Sig}} \uparrow\kappa' :\uparrow \mathtt{C}_T} \text{ (T-Jit-Restore)}$$

$$\frac{\Omega = \Omega', \Omega''_{\mathtt{ck}} \quad \mathtt{aID}(\mathtt{c}_0) \mid b \geq 0 : \mathtt{nat} \mid \Omega; \downarrow\Omega'' \vdash c_0 : \mathtt{C}_{\mathtt{unit}} \in \mathtt{Sig}}{\textsf{aID}(c_0) \mid b > 0 : \mathtt{nat} \mid \Omega \vdash_{\mathtt{Sig}} \uparrow\kappa' :\uparrow \mathtt{C}_{\mathtt{unit}}} \text{ (T-aID-Restore)}$$

Fig. 9. Crash, restore, and checkpoint typing

next power failure. Alternatively, in the JIT mode, checkpoints are dropped from Ω and execution continues with the expression or command κ, which was running right before the crash. In the atomic mode, execution continues with the original command c_0 enclosed in the atomic region. Instead of retyping the restored judgments, we check if there are already typing derivations by matching them up with the saved judgment in the signature.

5 Logical Relation for Intermittent Execution

We establish a logical relation to prove idempotency, which states that every intermittent execution of a program can be simulated by a continuous execution. The logical relation relates an intermittent execution with a continuous one and is indexed by Crash types. A continuous run is one with an infinite energy level, ∞. Crash types are recursive, yielding possible infinite atomic region re-executions. Thus, we use the maximum number of executions (also power failures) as a step index to stratify our logical relation to ensure its well-foundedness.

The logical relation (defined in Sec. 5.1) relies on `PwOff`, `Restore`, and `Commit` functions, referred to as power failure, restore, and commit policies, respectively. We establish specific policies for atomic and JIT execution modes. We formalize *semantic typing* as every atomic and JIT region of the program being logically-related to themselves. We prove that the semantically well-typed programs are idempotent across power failures in Sec. 5.2. The definitions match the memory operations in the dynamic rules that deal with crash, restore, and re-execution (D-S-aID/ D-S-Jit, D-R-aID/ D-R-Jit, and D-P-Ckpt/

$\texttt{Md} \mid b \geq 0 : \texttt{nat} \mid \Omega \mid \Sigma \Vdash c_1 \leq c_2 : \texttt{C}_{\texttt{unit}}$
iff $\forall n, m \geq 0.\ \forall \gamma, \mathsf{NV}, \mathsf{V}.s.t.\ \mathsf{NV} \mid \mathsf{V} \Vdash \gamma :: \Omega \mid \Sigma.$
$(\gamma \mid \texttt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c_1, \gamma \mid \texttt{Md} \mid \infty \mid \mathsf{NV} \mid \mathsf{V} \mid c_2) \in \mathcal{E}[\![\texttt{C}_{\texttt{unit}}]\!]^m$

Term Relation

$$\begin{aligned}
\mathcal{E}[\![\texttt{C}_{\texttt{unit}}]\!]^{m+1} = \{&(\gamma_1 \mid \texttt{Md} \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1, \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2)\ s.t.\\
&\exists.(\gamma_1' \mid \texttt{Md}' \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1')\ s.t.\\
&\gamma_1 \mid \texttt{Md} \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1 \rightarrow^*_{irred} \gamma_1' \mid \texttt{Md}' \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1' \wedge\\
&\exists.(\gamma_2' \mid \texttt{Md}' \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2')\ s.t.\\
&\quad \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2 \rightarrow^* \gamma_2' \mid \texttt{Md}' \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2' \wedge\\
&(\gamma_1' \mid \texttt{Md}' \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1', \gamma_2' \mid \texttt{Md}' \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2') \in \mathcal{V}[\![\texttt{C}_{\texttt{unit}}]\!]^{m+1}\}\\
\mathcal{E}[\![\texttt{C}_{\texttt{unit}}]\!]^{0} = \{&(\gamma_1 \mid \texttt{Md} \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1, \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2\ )\}
\end{aligned}$$

Value Relation

$$\begin{aligned}
\mathcal{V}[\![\uparrow\texttt{unit}]\!]^m &= \{(\gamma \,|\, \texttt{Md} \,|\, n_1 \,|\, \mathsf{NV}_1 \,|\, \mathsf{skip}, \gamma \,|\, \texttt{Md} \,|\, \infty \,|\, \mathsf{NV}_2 \,|\, \mathsf{skip})\ \mathsf{s.t.}\mathsf{NV}_1 = \mathsf{NV}_2\}\\
\mathcal{V}[\![\downarrow\uparrow\texttt{unit}]\!]^m &= \{(\gamma_1 \,|\, \texttt{Md} \,|\, n_1 \,|\, \mathsf{NV}_1 \,|\, \mathsf{V}_1 \,|\, \mathsf{skip}, \gamma_2 \,|\, \texttt{Md} \,|\, \infty \,|\, \mathsf{NV}_2 \,|\, \mathsf{V}_2 \,|\, \mathsf{skip})\ \mathsf{s.t.}\\
&\quad \texttt{Commit}(\gamma_i \,|\, \texttt{Md} \,|\, \mathsf{NV}_i \,|\, \mathsf{V}_i) = \gamma_1' \,|\, \mathsf{NV}_i' \wedge\\
&\quad (\gamma_1' \,|\, \texttt{Md} \,|\, n_1 \,|\, \mathsf{NV}_1' \,|\, \mathsf{skip}, \gamma_2 \,|\, \texttt{Md} \,|\, \infty \,|\, \mathsf{NV}_2' \,|\, \mathsf{skip}) \in \mathcal{V}[\![\uparrow\texttt{unit}]\!]^m\}\\
\mathcal{V}[\![\uparrow\texttt{C}_{\texttt{unit}}]\!]^m &= \{(\gamma_1 \,|\, \texttt{Md} \,|\, n \,|\, \mathsf{NV}_1 \,|\, \uparrow\kappa, \gamma_2 \,|\, \texttt{Md} \,|\, \infty \,|\, \mathsf{NV}_2 \,|\, \mathsf{V}_2 \,|\, c_2)\ \mathsf{s.t.}\\
&\quad \texttt{restore}(\gamma_1, \texttt{Md}, \mathsf{NV}_1, \kappa) = \mathsf{NV}_0 \,|\, \mathsf{V}_0 \,|\, c_0 \wedge\\
&\quad (\gamma_1 \,|\, \texttt{Md} \,|\, n \,|\, \mathsf{NV}_0 \,|\, \mathsf{V}_0 \,|\, c_0, \gamma_2 \,|\, \texttt{Md} \,|\, \infty \,|\, \mathsf{NV}_2 \,|\, \mathsf{V}_2 \,|\, c_2) \in \mathcal{E}[\![\texttt{C}_{\texttt{unit}}]\!]^m\}\\
\mathcal{V}[\![\texttt{nat}\rightsquigarrow\uparrow\texttt{C}_{\texttt{unit}}]\!]^m &= \{(\gamma_1 \,|\, \texttt{Md} \,|\, \cdot \,|\, \mathsf{NV}_1 \,|\, \varepsilon \ \texttt{\#}\ \mathsf{in}(n > 0, \uparrow\kappa), \gamma_2 \,|\, \texttt{Md} \,|\, \infty \,|\, \mathsf{NV}_2 | \mathsf{V}_2 | c_2)\ \mathsf{s.t.}\\
&\quad \forall n{>}0.(\gamma_1 \,|\, \texttt{Md} \,|\, n \,|\mathsf{NV}_1 |\, \uparrow\kappa, \gamma_2 \,|\, \texttt{Md} \,|\, \infty \,| \mathsf{NV}_2 | \mathsf{V}_2 |\, c_2) \in \mathcal{V}[\![\uparrow \texttt{C}_{\texttt{unit}}]\!]^m\}\\
\mathcal{V}[\![\downarrow(\texttt{nat}\rightsquigarrow\uparrow\texttt{C}_{\texttt{unit}})]\!]^m &= \{(\gamma_1 \,|\, \texttt{Md} \,|\, \cdot\ | \mathsf{NV}_1 | \mathsf{V}_1 |\, \downarrow\varepsilon\ \texttt{\#}\ \mathsf{in}(n > 0, \uparrow\kappa), \gamma_2 \,|\, \texttt{Md} \,|\, \infty \,| \mathsf{NV}_2 | \mathsf{V}_2 |\, c_2)\\
&\quad \mathsf{s.t.}\ \texttt{PwOff}(\gamma_1, \texttt{Md}, \mathsf{NV}_1, \mathsf{V}_1) = \gamma_1' \,|\, \mathsf{V}' \wedge\\
&\quad (\gamma_1' \,|\, \texttt{Md} \,|\, \cdot\ \,|\, \mathsf{V}', \mathsf{NV}_1 \,|\, \varepsilon\ \texttt{\#}\ \mathsf{in}(n > 0, \uparrow\kappa), \gamma_2 \,|\, \texttt{Md} \,|\, \infty \,|\, \mathsf{NV}_2 \,|\, \mathsf{V}_2 \,|\, c_2)\\
&\qquad\qquad \in \mathcal{V}[\![\texttt{nat} \rightsquigarrow \uparrow\texttt{C}_{\texttt{unit}}]\!]^m\}\\
\mathcal{V}[\![\texttt{C}_{\texttt{unit}}]\!]^{m+1} &= \{(\gamma_1 \,|\, \texttt{Md} \,|\, n_1 \,|\, \mathsf{NV}_1 \,|\, \mathsf{V}_1 \,|\, c_1,\ \gamma_2 \,|\, \texttt{Md} \,|\, \infty \,|\, \mathsf{NV}_2 \,|\, \mathsf{V}_2 \,|\, c_2)\\
&\quad \mathsf{s.t.}\ \texttt{either}\\
&\quad n_1 = 0 \wedge (\gamma_1 \,|\, \texttt{Md} \,|\, \cdot\ \,|\, \mathsf{NV}_1 \,|\, \mathsf{V}_1 \,|\, \downarrow\varepsilon\ \texttt{\#}\ \mathsf{in}(n_1 > 0, \uparrow c_1),\\
&\qquad\qquad \gamma_2 \,|\, \texttt{Md} \,|\, \infty \,|\, \mathsf{NV}_2 \,|\, \mathsf{V}_2 \,|\, c_2) \in\ \mathcal{V}[\![\downarrow(\texttt{nat} \rightsquigarrow\uparrow \texttt{C}_{\texttt{unit}})]\!]^m, \texttt{or}\\
&\quad n_1 > 0 \wedge (\gamma_1 \,|\, \texttt{Md} \,|\, n_1 \,|\, \mathsf{NV}_1 \,|\, \mathsf{V}_1 \,|\, c_1, \gamma_2 \,|\, \texttt{Md} \,|\, \infty \,|\, \mathsf{NV}_2 \,|\, \mathsf{V}_2 \,|\, c_2)\\
&\qquad\qquad \in \mathcal{V}[\![\downarrow\uparrow \texttt{unit}]\!]^m\}
\end{aligned}$$

Fig. 10. Logical relation

D-P-SEQ) for atomic and JIT regions, We prove that our syntactically well-typed programs are semantically well-typed. We generalize semantic typing rules, allowing custom power failure, restore, and commit policies (Sec. 5.3).

5.1 Semantic Typing via a Logical Relation

The logical relation, written $\texttt{Md} \mid b \geq 0 : \texttt{nat} \mid \Omega \mid \Sigma \Vdash c_1 \leq c_2 : \texttt{C}_{\texttt{unit}}$, is defined in Fig. 10 by a lexicographic induction on the index m and the structure of the types. The judgment $\mathsf{NV} \mid \mathsf{V} \Vdash \gamma :: \Omega \mid \Sigma$ in the definition states that γ maps the variables in Σ and Ω to locations in V and NV resp., such that their qualifiers and types match. Similar to prior work [2,16,42], our definition consists of a term relation $\mathcal{E}[\![\mathtt{C_{unit}}]\!]^m$ and a value relation $\mathcal{V}[\![\tau]\!]^m$.

Term Relation. A pair of open command configurations of type $\mathtt{C_{unit}}$ are in the term relation of index m if any intermittent execution of the first one after m power failures is indistinguishable from a continuous execution of the second one. In particular, for index $m+1$, the term relation relates two configurations at type $\mathtt{C_{unit}}$ if the first configuration eventually steps to a value (or "irreducible") configuration, i.e., it either evaluates to skip or its energy level depletes ($n_1' = 0$), and the second configuration can take zero or more steps such that the pair continue to be in the value relation of $\mathcal{V}[\![\mathtt{C_{unit}}]\!]^{m+1}$. When the index is $m = 0$, no execution is observed, so any two configurations are in the term relation. Here, *irred* refers to $\gamma_1' \mid \mathtt{Md}' \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1'$ being an irreducible configuration, i.e. it cannot take any more steps. Since our semantics for commands is deterministic, for each configuration $\gamma_1 \mid \mathtt{Md} \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1$ there is exactly one such irreducible configuration.

Value Relation. The value relation is defined based on the intended meaning of the type, and relates two value configurations that will have the same effect on the stores. The value relation relates two open command configurations at type $\mathtt{C_{unit}}$ and index $m+1$ if either (a) the first configuration has faced a power failure, and the two configurations continue to relate by $\mathcal{V}[\![\downarrow(\mathtt{nat} \rightsquigarrow \uparrow\mathtt{C_{unit}})]\!]^m$, or (b) the first configuration executed successfully without any power failures, and the two configurations are related by $\mathcal{V}[\![\downarrow\uparrow\mathtt{unit}]\!]^m$. This definition matches the disjunctive nature of type $\mathtt{C_{unit}}$, which is recursively defined in the signature as $\downarrow(\mathtt{nat} \rightsquigarrow \uparrow\mathtt{C_{unit}}) \vee \downarrow\uparrow\mathtt{unit}$. Since we unfold the recursive definition of $\mathtt{C_{unit}}$, we decrease the index from $m+1$ to m to ensure the relation's well-foundedness. Note that the value relation is neither defined nor called for $\mathtt{C_{unit}}$ at index 0.

The value relations in the third, fourth, and fifth rows of Fig. 10 are defined based on the type of the *first configuration*; the second configurations in these relations continue to be of type $\mathtt{C_{unit}}$. Only in the relations defined in the first and second rows of Fig. 10 do the types of both configurations match the indexed type of the relation. Hence, the value relation has varying arity: in the first and second rows of Fig. 10, the relation is *binary* while in the rest, the relation degenerates to *unary*, with the second configuration as its Kripke world [18].

The value relation at type $\downarrow(\mathtt{nat} \rightsquigarrow \uparrow\mathtt{C_{unit}})$ relates two configurations if the first one runs the crash instruction $\downarrow\varepsilon\ \mathtt{\#}\ \mathsf{in}(n > 0, \uparrow\kappa)$ and a power failure policy creates a checkpoint of volatile locations such that the configurations continue to be in the value relation at type $(\mathtt{nat} \rightsquigarrow \uparrow\mathtt{C_{unit}})$. The power failure function in an atomic mode is defined to checkpoint none of the volatile locations, i.e., $\mathtt{PwOff}(\gamma, \mathsf{aID}(c_0), \mathsf{NV}_1, \mathsf{V}_1) = \gamma' \mid \emptyset$, where γ' is the largest restriction of γ with $\mathit{range}(\gamma') = \mathsf{dom}(\mathsf{NV}_1)$, and defined to checkpoint all volatile locations in JIT mode, i.e., $\mathtt{PwOff}(\gamma, \mathsf{jit}, \mathsf{NV}_1, \mathsf{V}_1) = \gamma \mid \mathsf{V}_1$.

The value relation at type ($\mathtt{nat} \rightsquigarrow \uparrow\mathsf{C}_{\mathtt{unit}}$) is defined similarly to a function type in a value relation and requires the configurations to be related at type ($\uparrow\mathsf{C}_{\mathtt{unit}}$) for every energy input level n provided to the first configuration.

The value relation at type $\uparrow\mathsf{C}_{\mathtt{unit}}$ requires the first configuration to run the crash instruction $\uparrow\kappa$. The defined restore policy restores the nonvolatile memory NV_0, volatile memory V_0, and re-execution command c_0 such that the configurations continue to be related in the term interpretation at type $\mathsf{C}_{\mathtt{unit}}$. In an atomic mode, the restore function is defined as $\mathtt{restore}(\gamma, \mathsf{aID}(c), \mathsf{NV}_1, \kappa) = \mathsf{NV}_1 \mid \mathsf{NV}'' \mid c$ where $\mathsf{NV}_1 = \mathsf{NV}', \mathsf{NV}''_{\mathtt{ck}}$. In the JIT mode, the restore function is defined as $\mathtt{restore}(\gamma, \mathsf{jit}(c), \mathsf{NV}_1, \kappa) = \mathsf{NV}'_1 \mid \mathsf{NV}'' \mid c$ where $\mathsf{NV}_1 = \mathsf{NV}', \mathsf{NV}''_{\mathtt{ck}}$. We write $\mathsf{NV}_1 = \mathsf{NV}', \mathsf{NV}''_{\mathtt{ck}}$ to state that NV_1 can be uniquely partitioned into all locations ($\mathsf{NV}''_{\mathtt{ck}}$) that are checkpointed, i.e., of the form $\ell_{\mathtt{ck}}$, and regular locations (NV') of the form ℓ. NV'' is the non-checkpointed version of $\mathsf{NV}''_{\mathtt{ck}}$ which could be retrieved by removing the $\mathtt{ck}$ subscript from every location in $\mathsf{NV}''_{\mathtt{ck}}$.

The value relation at type $\downarrow\uparrow\mathtt{unit}$ requires both configurations to run skip, and the defined commit policy creates nonvolatile memories for both runs such that they continue to be related at type $\uparrow\mathtt{unit}$. In an atomic mode, the commit function is defined to replace the checkpointed locations in the nonvolatile memory with their volatile log, i.e., $\mathtt{Commit}(\gamma \mid \mathsf{aID}(c_0) \mid \mathsf{NV}_1 \mid \mathsf{V}_1) = \gamma' \mid \mathsf{NV}'_1 \mid \mathsf{V}''$, where $\mathsf{NV}_1 = \mathsf{NV}'_1, \mathsf{NV}''_{\mathtt{ck}}$ and $\mathsf{V}_1 = \mathsf{V}'_1, \mathsf{V}''$ and $\mathsf{dom}(\mathsf{V}'') = \mathsf{dom}(\mathsf{NV}'')$. Moreover, $\gamma' \subseteq \gamma$, with $\mathit{range}(\gamma') = \mathsf{dom}(\mathsf{NV}_1) \cup \mathsf{dom}(V'')$. In the JIT mode, the commit function simply drops all volatile memory, i.e., $\mathtt{Commit}(\gamma \mid \mathsf{jit} \mid \mathsf{NV}_1 \mid \mathsf{V}_1) = \gamma' \mid \mathsf{NV}_1$, $\gamma' \subseteq \gamma$, with $\mathsf{range}(\gamma') = \mathsf{dom}(\mathsf{NV}_1)$.

The value relation at type $\uparrow\mathtt{unit}$ requires the successful executions to store the same values in their memories, i.e., $\mathsf{NV}_1 = \mathsf{NV}_2$.

Semantic Typing. A program is semantically well-typed if every JIT and atomic region of it is self-related under our logical relation.

$$\frac{\mathsf{jit} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \cdot \Vdash c \leq c : \mathsf{C}_{\mathtt{unit}} \quad b : \mathtt{nat} \mid \Omega \Vdash p : \uparrow\mathsf{C}_{\mathtt{unit}}}{b : \mathtt{nat} \mid \Omega \Vdash c; p : \uparrow\mathsf{C}_{\mathtt{unit}}} \text{(P-SEQ-SEMANTIC)}$$

$$\frac{\begin{array}{c}\Omega_0 \mid \Sigma_0 = \mathsf{InitWorld}_t(\Omega; \rho) \\ \mathsf{aID}(c_0) \mid b \geq 0 : \mathtt{nat} \mid \Omega_0; \Sigma_0 \Vdash c_0 \leq c_0 : \mathsf{C}_{\mathtt{unit}} \quad b : \mathtt{nat} \mid \Omega \Vdash p : \uparrow\mathsf{C}_{\mathtt{unit}}\end{array}}{b : \mathtt{nat} \mid \Omega \Vdash \mathsf{Ckpt}[\mathsf{aID}, \rho](c_0); p : \uparrow\mathsf{C}_{\mathtt{unit}}} \text{(P-CKPT-SEMANTIC)}$$

5.2 Semantic Typing for Idempotency

The fundamental theorem of our logical relation states that syntactically well-typed programs are also semantically well-typed by proving that syntactically well-typed JIT and atomic regions are self-related. We state and prove the theorem in Sec. 6 but devote this section to explaining why being self-related implies idempotency. We explain it separately for JIT and atomic blocks.

Stepping a JIT block. Consider a program of form $[\chi_1 \rhd \varepsilon] \otimes \gamma_1 \mid n \mid \mathsf{NV}_1 \mid c_1; p$ that can take a step to $[\chi_k \rhd \varepsilon] \otimes \gamma \mid n'_k \mid \mathsf{NV}'_k \mid p$ via the D-P-SEQ rule. By the D-P-SEQ rule, we know that the command c_1 is successfully executed to completion with possibly m-many power failures along the way: $[\chi_1 \rhd \varepsilon] \otimes \gamma_1 \mid$

$\mathsf{jit} \mid n \mid \mathsf{NV}_1 \mid \cdot \mid c_1 \Rightarrow^* [\chi_k \rhd \varepsilon] \otimes \gamma'_k \mid \mathsf{jit} \mid n'_k \mid \mathsf{NV}'_k \mid \mathsf{V}'_k \mid \mathsf{skip}$. Our goal is to simulate this execution in a continuous setting. To model a continuous run, we run the configuration with ∞, an energy level: $[\chi \rhd \varepsilon] \otimes \gamma_1 \mid \mathsf{jit} \mid \infty \mid \mathsf{NV}_1 \mid \cdot \mid c_1 \Rightarrow^* [\chi \rhd \varepsilon] \otimes \gamma'_j \mid \mathsf{jit} \mid \infty \mid \mathsf{NV}'_j \mid \mathsf{V}'_j \mid \mathsf{skip}$.

Fig. 11 shows the construction of the simulation. We start with the assumption that the configuration with n energy level is self-related when given energy level ∞ for every index, including $m+1$ (point (1) in Fig. 11). We show that if the first configuration takes one or more steps, the second configuration can take zero or more steps so that the intermediate regions continue to relate.

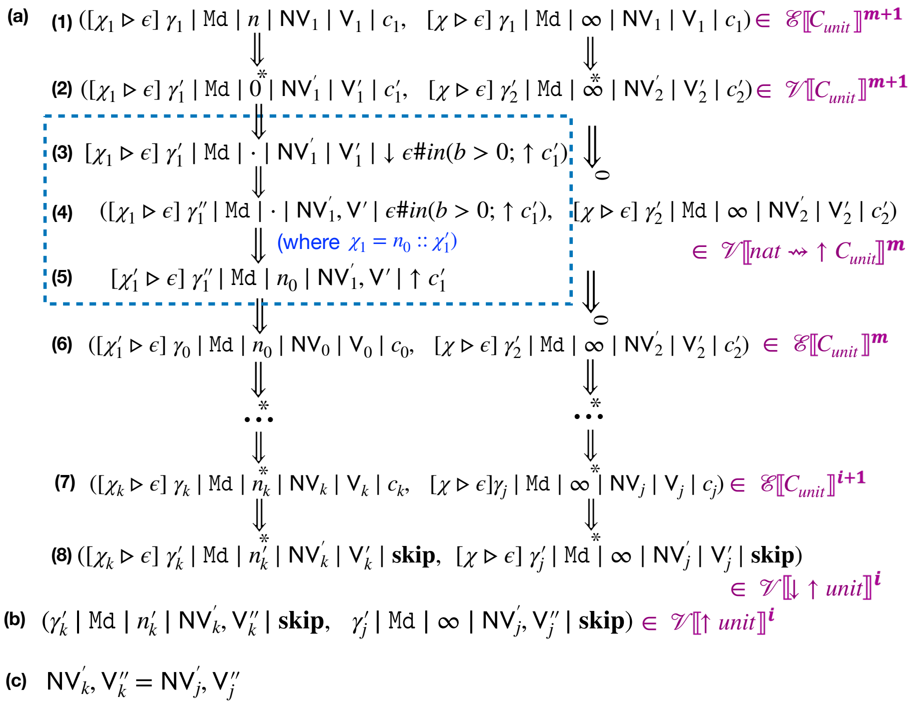
By definition of the term interpretation, c_1 in the first configuration is executed until the first power failure occurs. Moreover, by the relation, we can execute c_1 in the second configuration, too, such that the resulting configurations remain related (point (2) in Fig. 11) by the value interpretation at type $\mathsf{C}_{\mathtt{unit}}$. The first configuration takes a step from point (2) to point (3) using the D-CRASH rule by the computational semantics. By the definition of the logical relation, the two configurations continue to be related by the value interpretation at type $\downarrow(\mathtt{nat} \rightsquigarrow \uparrow \mathsf{C}_{\mathtt{unit}})$. Then the first configuration takes a step from point (3) to point (4) by the D-S-JIT rule; in this case, we know (by the assumptions of the rule) $\mathsf{V}' = \mathsf{V}'_1$ and $\gamma''_1 = \gamma$. This matches the definition of the power-off policy for JIT blocks (see Sec. 5.1), and thus the two configurations remain related by the value relation at type $\mathtt{nat} \rightsquigarrow \uparrow \mathsf{C}_{\mathtt{unit}}$. Next, the first configuration takes a step to point (5) by inputting a new energy level from the environment (n_2). By the definition of the value relations, the two configurations will remain related by the value interpretation at type $\uparrow \mathsf{C}_{\mathtt{unit}}$.

Finally, the configuration steps to point (6) by D-RESTORE-JIT that copies all checkpointed locations inside the volatile memory and continues by running the interrupted command κ, i.e., here $\mathsf{NV}_0 = \mathsf{NV}'_1$ and $\mathsf{V}_0 = \mathsf{V}' = \mathsf{V}'_1$ and $c_0 = \kappa$. This matches the restore policy defined for JIT regions; thus, the configurations continue to be related by the *term relation* at type $\mathsf{C}_{\mathtt{unit}}$, similar to what we had earlier at point (1) in Fig. 11, but with fewer power failures remaining.

Now, when the first configuration finally steps to point (8), by the definition of the logical relation, we know that the second configuration steps into skip too. Thus, we can apply the D-Ckpt rule on the second configuration. The volatile memory V'_j is dropped, and the mapping is reset to γ, i.e., it matches the commit policy defined for JIT blocks. in the logical relation. By Fig. 11-d, we get $\mathsf{NV}'_j = \mathsf{NV}'_k$, which completes deriving our goal.

Stepping an atomic region. We can build the desired simulation by taking the same steps described for a JIT region. Similarly, the key point is that the power-off and restore policies exactly match how the rules D-S-AID and D-RESTORE-AID, respectively, handle nonvolatile and volatile memories, and the commit policy corresponds to the $\mathsf{FinWorld}$ function in the D-CKPT rule.

We showed that our logical relation ensures idempotency for JIT and atomic regions. In the next section, we show that our logical relation formalizes a semantic typing to ensure idempotency of more general policies.



(a)
(1) $([\chi_1 \rhd \epsilon]\ \gamma_1 \mid \mathtt{Md} \mid n \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1,\ \ [\chi \rhd \epsilon]\ \gamma_1 \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1) \in \mathscr{E}[\![C_{unit}]\!]^{m+1}$
(2) $([\chi_1 \rhd \epsilon]\ \gamma_1' \mid \mathtt{Md} \mid 0 \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1',\ \ [\chi \rhd \epsilon]\ \gamma_2' \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2') \in \mathscr{V}[\![C_{unit}]\!]^{m+1}$
(3) $[\chi_1 \rhd \epsilon]\ \gamma_1' \mid \mathtt{Md} \mid \cdot \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid \downarrow \epsilon\#in(b > 0; \uparrow c_1')$
(4) $([\chi_1 \rhd \epsilon]\ \gamma_1'' \mid \mathtt{Md} \mid \cdot \mid \mathsf{NV}_1', \mathsf{V}' \mid \epsilon\#in(b > 0; \uparrow c_1'),\ \ [\chi \rhd \epsilon]\ \gamma_2' \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2')$
(where $\chi_1 = n_0 :: \chi_1'$)
$\in \mathscr{V}[\![nat \rightsquigarrow \uparrow C_{unit}]\!]^{m}$
(5) $[\chi_1' \rhd \epsilon]\ \gamma_1'' \mid \mathtt{Md} \mid n_0 \mid \mathsf{NV}_1', \mathsf{V}' \mid \uparrow c_1'$
(6) $([\chi_1' \rhd \epsilon]\ \gamma_0 \mid \mathtt{Md} \mid n_0 \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0,\ \ [\chi \rhd \epsilon]\ \gamma_2' \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2') \in \mathscr{E}[\![C_{unit}]\!]^{m}$
(7) $([\chi_k \rhd \epsilon]\ \gamma_k \mid \mathtt{Md} \mid n_k \mid \mathsf{NV}_k \mid \mathsf{V}_k \mid c_k,\ \ [\chi \rhd \epsilon]\gamma_j \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_j \mid \mathsf{V}_j \mid c_j) \in \mathscr{E}[\![C_{unit}]\!]^{i+1}$
(8) $([\chi_k \rhd \epsilon]\ \gamma_k' \mid \mathtt{Md} \mid n_k' \mid \mathsf{NV}_k' \mid \mathsf{V}_k' \mid \mathbf{skip},\ \ [\chi \rhd \epsilon]\ \gamma_j' \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_j' \mid \mathsf{V}_j' \mid \mathbf{skip})$
$\in \mathscr{V}[\![\downarrow \uparrow unit]\!]^{i}$
(b) $(\gamma_k' \mid \mathtt{Md} \mid n_k' \mid \mathsf{NV}_k', \mathsf{V}_k'' \mid \mathbf{skip},\ \ \gamma_j' \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_j', \mathsf{V}_j'' \mid \mathbf{skip}) \in \mathscr{V}[\![\uparrow unit]\!]^{i}$
(c) $\mathsf{NV}_k', \mathsf{V}_k'' = \mathsf{NV}_j', \mathsf{V}_j''$

Fig. 11. Why the logical relation is enough.

5.3 More General Policies

We utilize our semantic typing to allow custom policies for power failure, restore, and commit. We extend the grammar of programs as $p := \cdot \mid \mathsf{Reg}[\mathsf{aID}, \overrightarrow{arg}](c); p$, where $\overrightarrow{arg}$ refers to the arguments that the programmer decides to pass to the region for initialization. To each region, we assign a unique identifier aID that is associated with the three policies and two functions $\texttt{InitGeneral}_t$ and $\texttt{InitGeneral}_d$ to initialize the static and dynamic memories, respectively. We add the following semantic typing rule for the general regions:

$$\frac{\begin{array}{c} c_0 \mid \Omega_0 \mid \Sigma_0 = \texttt{InitGeneral}_t(\Omega; \mathsf{aID}; c; \overrightarrow{arg}) \\ \mathsf{aID}(c_0) \mid b \geq 0 : \texttt{nat} \mid \Omega_0; \Sigma_0 \Vdash c_0 \leq c_0 : \mathtt{C}_{\mathtt{unit}} \quad b : \texttt{nat} \mid \Omega \Vdash p : \uparrow\mathtt{C}_{\mathtt{unit}} \end{array}}{b : \texttt{nat} \mid \Omega \Vdash \mathsf{Reg}[\mathsf{aID}, \overrightarrow{arg}](c); p : \uparrow\mathtt{C}_{\mathtt{unit}}} \text{(P-Reg-semantic)}$$

For a self-related region to be idempotent, its policies `Commit`, `PwOff`, and `Restore` must match the dynamics, so we add dynamic rules for custom regions in Fig. 12. The JIT and atomic region policies and their dynamic rules are instances of these general policies. As an example, the programmer can customize the policies of the first block of Fig. 1 to not checkpoint variable u. The program remains idempotent as the atomic region never reads u before writing to it. This

$$\frac{\gamma_0 \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0 = \mathsf{restore}(\mathsf{NV}, \mathsf{V}, \kappa, \mathtt{Md}, \gamma)}{[\chi \rhd \varepsilon] \otimes \gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid\uparrow \kappa \;\Rightarrow\; [\chi \rhd \varepsilon] \otimes \gamma_0 \mid \mathtt{Md} \mid n \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0} \text{ (D-R-Reg)}$$

$$\frac{\begin{array}{c} n > 0 \quad \mathtt{InitGeneral}_d(\mathsf{NV}; \mathsf{aID}; c; \gamma; \overrightarrow{arg}) = c_0, \mathsf{NV}_0, \mathsf{V}_0 \\ [\chi \rhd \varepsilon] \otimes \mathsf{aID}(c_0) \mid n \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0 \Rightarrow^* [\chi' \rhd \varepsilon] \otimes \mathsf{aID}(c_0) \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid \mathsf{skip} \\ n' > 0 \quad \mathsf{NV}_1 = \mathsf{Commit}(\mathsf{NV}'; \mathsf{V}'; \mathsf{aID}; \overrightarrow{arg}) \end{array}}{[\chi \rhd \varepsilon] \otimes \gamma \mid n \mid \mathsf{NV} \mid \mathsf{Reg}[(\mathsf{aID}; \overrightarrow{arg})](c); p \;\Rightarrow\; [\chi' \rhd \varepsilon] \otimes \gamma \mid n' \mid \mathsf{NV}_1 \mid p} \text{ (D-Reg)}$$

$$\frac{\mathsf{V}' = \mathtt{PwOff}(\mathsf{NV}, \mathsf{V}, \mathtt{Md}, \gamma)}{\begin{array}{c}[\chi \rhd \varepsilon] \otimes \gamma \mid \mathtt{Md} \mid \cdot \mid \mathsf{NV} \mid \mathsf{V} \mid \downarrow\varepsilon \;\#\; \mathsf{in}(b > 0; \uparrow\kappa) \;\Rightarrow \\ [\chi \rhd \varepsilon] \otimes \gamma \mid \mathtt{Md} \mid \cdot \mid \mathsf{NV}, \mathsf{V}' \mid \varepsilon \;\#\; \mathsf{in}(b > 0; \uparrow\kappa)\end{array}} \text{ (D-S-Reg)}$$

Fig. 12. Custom dynamic rules

policy is implemented by real systems [23,24,41]. Our static typing rules can be extended to reason about them as shown in the companion technical report.

6 Metatheory

This section establishes the main properties of the system, which are progress and preservation, adequacy, and the most important result: the fundamental theorem where we prove that statically well-typed programs are semantically well-typed. The theorems and their complete proofs are provided in the companion TR [15].

The progress and preservation theorems assume memory locations to be well-formed, $\vdash^{\mathtt{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$, which is defined similarly to the $\mathsf{NV} \mid \mathsf{V} \Vdash \gamma : \Omega \mid \Sigma$ used in the logical relation, but imposes extra conditions based on the execution mode $\mathtt{Md}$. It states that γ maps variables in contexts Ω and Σ to the nonvolatile and volatile memories, NV and V, respectively, such that their qualifiers and the type of the stored values match. Moreover, it requires specific properties on the contexts depending on $\mathtt{Md}$; in atomic mode, each checkpointed location in NV and Ω must have copies in V and Σ. We state the theorems below.

Theorem 1 (Progress for Commands). *If* $\mathtt{Md} \mid b\, \mathcal{R}\, m : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \tau$, *then* $\forall\, n : \mathtt{nat}$ *with* $n \mathcal{R} m$ *and* $\forall\, \gamma, \mathsf{NV}, \mathsf{V}$ *with* $\vdash^{\mathtt{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$, *either* $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c$ *is a value, or for some configuration* $\gamma' \mid \mathtt{Md}' \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'$ *we have* $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c \;\rightarrow\; \gamma' \mid \mathtt{Md}' \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'$. *Moreover, if* $\mathtt{Md}$ *is an atomic mode, we have* $\mathsf{NV}' = \mathsf{NV}$.

Theorem 2 (Preservation for Commands). *If* $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \tau$, *and for some* $\vdash^{\mathtt{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$ *and* $n{:}\mathtt{nat} \geq 0$, *we have* $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c \;\rightarrow\; \gamma' \mid \mathtt{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'$, *then for some* Σ_1, *we have* $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma_1 \vdash_{\mathtt{Sig}} c' : \tau$, *where* $\vdash^{\mathtt{Md}}_{\gamma'} \mathsf{NV}' \mid \mathsf{V}' : \Omega \mid \Sigma_1$ *and* $n' \geq 0$.

Theorem 3 (Fundamental Theorem). *If* $b : \mathtt{nat} \mid \Omega \vdash p : \uparrow\mathsf{C}_{\mathtt{unit}}$, *then* $b : \mathtt{nat} \mid \Omega \Vdash p : \uparrow\mathsf{C}_{\mathtt{unit}}$.

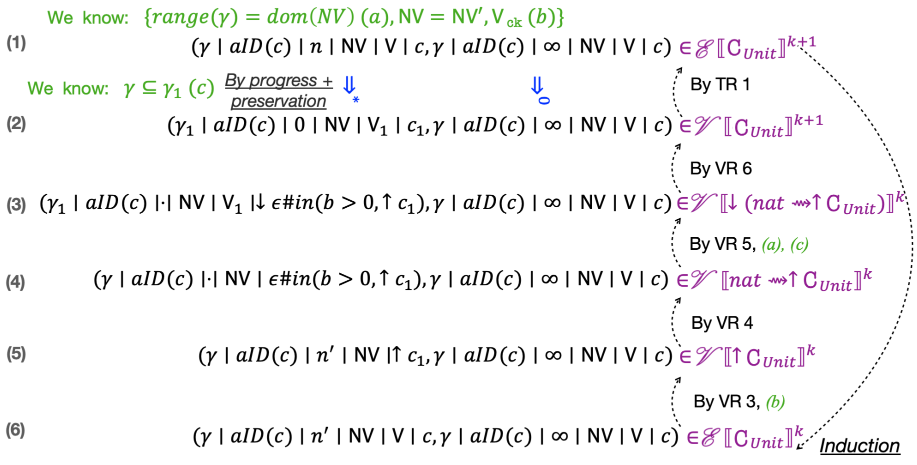

We know: {range(γ) = dom(NV) (a), NV = NV′, V_ck (b)}
(1) (γ | aID(c) | n | NV | V | c, γ | aID(c) | ∞ | NV | V | c) ∈ ℰ⟦C_Unit⟧^{k+1}
We know: γ ⊆ γ_1 (c)
By progress + preservation
By TR 1
(2) (γ_1 | aID(c) | 0 | NV | V_1 | c_1, γ | aID(c) | ∞ | NV | V | c) ∈ 𝒱⟦C_Unit⟧^{k+1}
By VR 6
(3) (γ_1 | aID(c) |·| NV | V_1 |↓ ε#in(b > 0, ↑ c_1), γ | aID(c) | ∞ | NV | V | c) ∈ 𝒱⟦↓ (nat ⇝ ↑ C_Unit)⟧^k
By VR 5, (a), (c)
(4) (γ | aID(c) |·| NV | ε#in(b > 0, ↑ c_1), γ | aID(c) | ∞ | NV | V | c) ∈ 𝒱⟦nat ⇝ ↑ C_Unit⟧^k
By VR 4
(5) (γ | aID(c) | n′ | NV |↑ c_1, γ | aID(c) | ∞ | NV | V | c) ∈ 𝒱⟦↑ C_Unit⟧^k
By VR 3, (b)
(6) (γ | aID(c) | n′ | NV | V | c, γ | aID(c) | ∞ | NV | V | c) ∈ ℰ⟦C_Unit⟧^k
Induction

Fig. 13. Proof of the fundamental theorem for P-Ckpt

The proof of Theorem 3 is by induction on the static typing derivation for p and considers the last step in the derivation. Fig. 13 explains the idea of the proof for the case where P-Ckpt is the last step of the derivation. By inversion, $p = \mathsf{Ckpt}[\mathsf{aID}, \rho](c); p'$. Also, c is well-typed for static contexts Ω' and Σ, where $\Omega' = \Omega'', \Sigma_{\mathtt{ck}}$. The goal is to establish point (1) in the figure: c is related to itself in the term interpretation for arbitrary n, m, γ, NV and V where $\mathsf{NV} \mid \mathsf{V} \Vdash \gamma{::}\Omega'', \Sigma_{\mathtt{ck}} \mid \Sigma$. The last condition enforces that the static contexts match the dynamic context. The condition also establishes the more refined well-formedness condition that $\vdash^{\mathtt{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$ in atomic mode, required by progress and preservation, since it enforces that each checkpointed location in NV and Ω have copies in V and Σ. In particular, $\mathsf{NV} = \mathsf{NV}', \mathsf{V}_{\mathtt{ck}}$ and $\mathit{range}(\gamma) = \mathit{dom}(\mathsf{NV})$. When $m = 0$, the proof is trivial. Consider the case where $m = k + 1$. By the progress and preservation theorems, the first configuration can take multiple steps until it becomes a value $\gamma_1 \mid \mathsf{aID}(c) \mid n' \mid \mathsf{NV} \mid \mathsf{V}_1 \mid c_1$ that continues to be well-typed. If $n' > 0$, the second configuration steps similarly to completion and establishes that the two resulting configurations are in the value relation. This case is not shown in the figure. If $n' = 0$, the second configuration does not step and instead reaches point (2) in Fig. 13. At point (2), the proof must show that the configurations are in the value interpretation at type $\mathsf{C}_{\mathtt{unit}}$.

The dashed line in the figure states that establishing point (2) implies the relation in point (1). The cascade of implications (dashed lines) follows the definition of the value relations at each type. At each step, we invert on the typing rule of the open configuration and show that runtime memories stay well-defined for static contexts. At point (4), we apply the power failure policy for atomic regions, which drops the volatile memory V_1 and creates a mapping using the domain of NV. By the prior conditions established, we know the created mapping is the original mapping γ. At point (6), we apply the restore policy for atomic regions, which creates a new volatile memory based on NV. Again by the prior conditions established, we know the volatile memory created is the original volatile V. The goal at point (6) is similar to our original goal at point (1), except that the proof uses an inductive argument to relate the two configurations at k.

Finally the Adequacy Theorem states that semantically well-typed programs are idempotent, defined below. The proof is illustrated in Section 5.2.

Definition 1 (Idempotency). *A triple of a program p, nonvolatile memory NV, and a mapping γ is idempotent, if every intermittent execution of the program can be simulated by a continuous execution of it: for all $n, n', \chi_1, \chi_1', \mathsf{NV}', p'$, if $[\chi_1 \rhd \varepsilon] \otimes \gamma \mid n \mid \mathsf{NV} \mid p \Rightarrow [\chi_1' \rhd \varepsilon] \otimes \gamma \mid n' \mid \mathsf{NV}' \mid p'$, then $[\chi_2 \rhd \varepsilon] \otimes \gamma \mid \infty \mid \mathsf{NV} \mid p \Rightarrow [\chi_2 \rhd \varepsilon] \otimes \gamma \mid \infty \mid \mathsf{NV}' \mid p'$.*

Theorem 4 (Adequacy). *Consider $b : \mathtt{nat} \mid \Omega \Vdash p : \mathsf{C}_{\mathtt{unit}}$, a nonvolatile memory NV and a bijective map γ that matches qualifiers and types from variables in Ω to locations in NV. The triple of p, NV, and γ is idempotent.*

7 Discussion & Related Work

Intermittent Computing. Surbatovich et al. [41] provide the first formal framework for reasoning about intermittent execution, give the correctness definition that we use, and identify precise memory invariants needed for an execution to be correct. Our Crash types capture some of these invariants; capturing all requires reasoning about the effects of non-deterministic sensor inputs, which we leave to future work. This work is the first to treat intermittent operations at the type level and explore the logical interpretation of intermittent execution. We speculate that our type-based approach using logical relations will provide a cleaner foundation for reasoning about the correctness of more complex intermittent systems, e.g., concurrent ones. Other works that investigate the formal properties of intermittent computing either reason about the effects of intermittent execution on peripheral interactions [9] or enforce timeliness constraints on sensor readings [40], which are orthogonal to ours.

Adjoint Logic. Benton et al. [7,8] provided the first categorical foundation for using adjoint functors to combine linear and nonlinear logics and showed that a well-behaved calculus requires an independence principle: linear formulae cannot appear in the assumptions of a nonlinear sequent. Follow up works further generalized the system [20,21,36]. There, the relation to Pfenning and Davies's [30] formulation of the lax $\bigcirc$ modality was noted; $\bigcirc$ corresponds to UF, where F and U are adjunctions between truth and validity categories. Short of a full curry-howard correspondence for our type system and underlying logic, we designed the rules for $\uparrow$ and $\downarrow$ based on the above calculi. Our stable and unstable contexts correspond to the validity and truth contexts respectively. Thus, we speculate that the combination $\uparrow\downarrow$ in our system corresponds to the lax modality.

Several prior works used type systems with adjoint modalities to model switching between program modes [6,14,34], e.g., switching a processes' mode between shared and unshared [6], or adding multicasting, replicable services, and cancellation modes to a session-typed message passing system [34]. We are the first to use these modalities to handle unforeseen shut-downs and distinguish between stable and power-failure prone modes.

Logical Relations. Prior work [3,42] uses step indexing to ensure the well-foundedness of logical relations that handle heaps with cyclic references, dynamic memory allocation, or recursive types. Our Crash types model the infinite computation that an atomic region can experience under a non-deterministic number of power failures and re-executions. This recursion necessitates an-indexed relation that limits the number of execution attempts a program can make.

Jung and Tiuryn introduced a logical relation for lambda definability that allows varying arities [18]. The idea is to increase the arity when passing to later worlds instead of starting with a large arity. Our logical relation can also be viewed as a relation with different arities; the initial type of the relation is binary, while after a crash the type of the value relation only corresponds to the intermittent configuration. During these value steps, the relation is unary, with the continuous configuration acting as a kripke world for the intermittent configuration. After restoration, the relation reverts to binary.

Logical relations have been widely used to prove program equivalence, e.g., [2,3,10,16]. At a high level, idempotency is similar to program equivalence, but it handles re-execution and requires us only to prove simulation from an intermittent to continuous run, not vice-versa.

Algebraic Effect Handlers. Algebraic effect handlers [27,31,32,33] give a unified theory for computational effects, e.g., exceptions and interactive input/output. A handler accesses the continuation to transform the computation. Following effect handler syntax, we write effectful environmental interactions of our system as $\varepsilon\#\mathsf{in}(b > 0, \uparrow\kappa)$, where b refers to a natural number returned by the environment and $\uparrow\kappa$ is the continuation. Our restore policy resembles a handler, in that it has access to the continuation, but an atomic region may dismiss the continuation, restarting from a saved command.

Crash Hoare Logic. Crash Hoare logic (CHL) [11] ensures the correctness of crash and restore operations in a file system. CHL extends Hoare logic with a crash condition and a recovery procedure. The crash condition states what happens to the state on a crash. The recovery procedure runs after the crash and manipulates the state before resuming. The system checks that if the program crashes, the storage system will recover to a state consistent with the specifications. Unlike us, they do not care about idempotency, requiring manual effort to formalize the crash condition and recovery policy. Our syntactic typing fixes the power failure, restore, and commit policies, and our formal results guarantee that following the policies ensures idempotency, the common correctness condition for intermittent execution. We also allow the programmer to formalize bespoke semantically well-typed policies.

8 Conclusion

This work provides the first logical interpretation of intermittent execution. It shows that adjoint logic can be applied to define Crash types, which internalize the dualities between stable and unstable values, and complete versus partial (re-)executions of intermittent programs. The typing constraints capture invariants of power failure, restoration, and re-execution in intermittent systems. The proofs of progress, preservation, and the fundamental theorem imply the correctness of intermittent systems, i.e. idempotency of execution.

References

1. Adkins, J., Campbell, B., Ghena, B., Jackson, N., Pannuto, P., Dutta, P.: The signpost network: Demo abstract. In: Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM. SenSys '16 (2016). https://doi.org/10.1145/2994551.2996542
2. Ahmed, A., Dreyer, D., Rossberg, A.: State-dependent representation independence. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 340–353. POPL '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1480881.1480925
3. Ahmed, A.J.: Semantics of types for mutable state. Princeton University (2004)
4. Balsamo, D., Weddell, A., Das, A., Arreola, A., Brunelli, D., Al-Hashimi, B., Merrett, G., Benini, L.: Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **PP**(99), 1–1 (2016). https://doi.org/10.1109/TCAD.2016.2547919
5. Balsamo, D., Weddell, A.S., Merrett, G.V., Al-Hashimi, B.M., Brunelli, D., Benini, L.: Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. IEEE Embedded Systems Letters **7**(1), 15–18 (2015). https://doi.org/10.1109/LES.2014.2371494
6. Balzer, S., Toninho, B., Pfenning, F.: Manifest deadlock-freedom for shared session types. In: Proceedings of the 29th European Symposium on Programming. pp. 611–639 (2019)
7. Benton, N., Wadler, P.: Linear logic, monads and the lambda calculus. In: Proceedings 11th Annual IEEE Symposium on Logic in Computer Science. pp. 420–431. IEEE (1996)
8. Benton, P.N.: A mixed linear and non-linear logic: Proofs, terms and models. In: International Workshop on Computer Science Logic. pp. 121–135. Springer (1994)
9. Berthou, G., Dagand, P.E., Demange, D., Oudin, R., Risset, T.: Intermittent computing with peripherals, formally verified. In: The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. pp. 85–96. LCTES '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372799.3394365
10. Birkedal, L., Støvring, K., Thamsborg, J.: Realizability semantics of parametric polymorphism, general references, and recursive types. In: International Conference on Foundations of Software Science and Computational Structures. pp. 456–470. FOSSACS '09, Springer (2009)
11. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N.: Using crash hoare logic for certifying the fscq file system. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 18–37. SOSP '15 (2015)

12. Colin, A., Lucia, B.: Chain: Tasks and channels for reliable intermittent programs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA '16 (2016). https://doi.org/10.1145/2983990.2983995
13. Dahiya, M., Bansal, S.: Automatic verification of intermittent systems. In: Dillig, I., Palsberg, J. (eds.) Verification, Model Checking, and Abstract Interpretation. VMCAI '18 (2018)
14. Das, A., Balzer, S., Hoffmann, J., Pfenning, F., Santurkar, I.: Resource-aware session types for digital contracts. In: IEEE 34th Computer Security Foundations Symposium. pp. 1–16. CSF '21 (2021)
15. Derakhshan, F., Dotzel, M., Surbatovich, M., Jia, L.: Technical report: Modal crash types for intermittent computing. Tech. rep., Carnegie Mellon University (2023). https://doi.org/10.1184/R1/21950804
16. Dreyer, D., Neis, G., Birkedal, L.: The impact of higher-order state and control effects on local relational reasoning. Journal of Functional Programming **22**(4-5), 477–528 (2012)
17. Hester, J., Storer, K., Sorber, J.: Timely execution on intermittently powered batteryless sensors. In: Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (2017). https://doi.org/10.1145/3131672.3131673
18. Jung, A., Tiuryn, J.: A new characterization of lambda definability. In: International Conference on Typed Lambda Calculi and Applications. pp. 245–257. Springer (1993)
19. Kortbeek, V., Yildirim, K.S., Bakar, A., Sorber, J., Hester, J., Pawełczak, P.: Time-sensitive intermittent computing meets legacy software. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 85–99. ASPLOS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3373376.3378476
20. Licata, D.R., Shulman, M.: Adjoint logic with a 2-category of modes. In: International Symposium on Logical Foundations of Computer Science. pp. 219–235. Springer (2016)
21. Licata, D.R., Shulman, M., Riley, M.: A fibrational framework for substructural and modal logics. In: 2nd International Conference on Formal Structures for Computation and Deduction. FSCD '17, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
22. Lucia, B., Denby, B., Manchester, Z., Desai, H., Ruppel, E., Colin, A.: Computational nanosatellite constellations: Opportunities and challenges. GetMobile: Mobile Comp. and Comm. **25**(1), 16–23 (Jun 2021). https://doi.org/10.1145/3471440.3471446
23. Lucia, B., Ransford, B.: A simpler, safer programming and execution model for intermittent systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '15 (2015). https://doi.org/10.1145/2737924.2737978
24. Maeng, K., Colin, A., Lucia, B.: Alpaca: Intermittent execution without checkpoints. Proc. ACM Program. Lang. **1**(OOPSLA), 96:1–96:30 (Oct 2017). https://doi.org/10.1145/3133920
25. Maeng, K., Lucia, B.: Supporting peripherals in intermittent systems with just-in-time checkpoints. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1101–1116. PLDI '19 (2019). https://doi.org/10.1145/3314221.3314613

26. Maeng, K., Lucia, B.: Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 1005–1021. PLDI '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3385998
27. Moggi, E.: Computational lambda-calculus and monads. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science (1988)
28. Nardello, M., Desai, H., Brunelli, D., Lucia, B.: Camaroptera: A batteryless long-range remote visual sensing system. In: Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems. pp. 8–14. ENSsys'19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3362053.3363491
29. NASA: What is KickSat-2? `https://www.nasa.gov/ames/kicksat` (2019), visited April 15th, 2022
30. Pfenning, F., Davies, R.: A judgmental reconstruction of modal logic. Mathematical structures in computer science **11**(4), 511–540 (2001)
31. Plotkin, G., Power, J.: Semantics for algebraic operations. Electronic Notes in Theoretical Computer Science **45**, 332–345 (2001)
32. Plotkin, G., Pretnar, M.: Handlers of algebraic effects. In: Proceedings of the 19th European Symposium on Programming. pp. 80–94. Springer (2009)
33. Pretnar, M., Plotkin, G.D.: Handling algebraic effects. Logical methods in computer science **9** (2013)
34. Pruiksma, K., Pfenning, F.: A message-passing interpretation of adjoint logic. Journal of Logical and Algebraic Methods in Programming **120**, 100637 (2021)
35. Ransford, B., Sorber, J., Fu, K.: Mementos: System support for long-running computation on RFID-scale devices. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XVI (2011). https://doi.org/10.1145/1950365.1950386
36. Reed, J.: A judgmental deconstruction of modal logic. Unpublished manuscript, January (2009)
37. Ruppel, E., Lucia, B.: Transactional concurrency control for intermittent, energy-harvesting computing systems. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1085–1100. PLDI '19 (2019). https://doi.org/10.1145/3314221.3314583
38. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing. pp. 204–213. PODC '95 (1995). https://doi.org/10.1145/224964.224987
39. Surbatovich, M., Jia, L., Lucia, B.: I/o dependent idempotence bugs in intermittent systems. Proc. ACM Program. Lang. **3**(OOPSLA), 183:1–183:31 (Oct 2019). https://doi.org/10.1145/3360609
40. Surbatovich, M., Jia, L., Lucia, B.: Automatically enforcing fresh and consistent inputs in intermittent systems. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 851–866. PLDI '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454081
41. Surbatovich, M., Lucia, B., Jia, L.: Towards a formal foundation of intermittent computing. Proc. ACM Program. Lang. **4**(OOPSLA) (Nov 2020). https://doi.org/10.1145/3428231
42. Thamsborg, J., Birkedal, L.: A kripke logical relation for effect-based program transformations. ACM SIGPLAN Notices **46**(9), 445–456 (2011)

43. Van Der Woude, J., Hicks, M.: Intermittent computation without hardware support or programmer intervention. In: Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation (2016). https://doi.org/10.5555/3026877.3026880
44. Yildirim, K.S., Majid, A.Y., Patoukas, D., Schaper, K., Pawelczak, P., Hester, J.: Ink: Reactive kernel for tiny batteryless sensors. In: Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems. pp. 41–53. SenSys '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3274783.3274837

Gradual Tensor Shape Checking

Momoko Hattori(✉), Naoki Kobayashi(✉), and Ryosuke Sato(✉)

The University of Tokyo, Tokyo, Japan
{momohatt,koba,rsato}@is.s.u-tokyo.ac.jp

Abstract. Tensor shape mismatch is a common source of bugs in deep learning programs. We propose a new type-based approach to detect tensor shape mismatches. One of the main features of our approach is the best-effort shape inference. As the tensor shape inference problem is undecidable in general, we allow static type/shape inference to be performed only in a best-effort manner. If the static inference cannot guarantee the absence of the shape inconsistencies, dynamic checks are inserted into the program. Another main feature is gradual typing, where users can improve the precision of the inference by adding appropriate type annotations to the program. We formalize our approach and prove that it satisfies the criteria of gradual typing proposed by Siek et al. in 2015. We have implemented a prototype shape checking tool based on our approach and evaluated its effectiveness by applying it to some deep neural network programs.

1 Introduction

Tensor Shape Checking and Its Difficulties. Tensor shape mismatch is one of the common sources of dynamic errors in programs using tensors (i.e., multi-dimensional arrays). For example, the reshape operation of tensors takes a tensor x and an integer list S and returns a new tensor of the shape S obtained by realigning the elements in x. The input and output tensors must have the same number of elements; a tensor of shape $[2; 3; 4]$[1] can be reshaped into a shape $[3; 2; 4]$, while trying to reshape it into $[3; 4]$ results in a runtime error.

Early detection of tensor shape mismatch errors is critical in particular for deep learning programs, where tensors are frequently used. Since deep learning programs often take a considerable amount of time to train networks, it is often the case that a program takes hours and days to compute the weights of deep neural networks only to be terminated by one tensor shape mismatch error, throwing away the trained weights. Even worse, some tensor shape mismatches can be harder to notice: mixing up the height and the width of square images does not raise runtime errors but degrades the performance of the neural network.

The existing work on static detection of tensor shape mismatch errors can be classified into two categories. One is the whole-program analysis approach [17,31], which collects tensor shape information by partially evaluating

[1] In this paper, we denote lists in the OCaml-style as in $[1; 2; 3]$ to disambiguate it from the citations.

```
let model s =
  let f = ... in let g = ... in fun x -> let y = f x in g y
let _ = model 1 (Tensor.rand [20])
```

Fig. 1. An OCaml program written with OCaml-Torch.

the program in the style of abstract interpretation. The other is the type-based approach [3,25], which expresses the shapes of tensors as a part of the type information. Still, none of them is fully satisfactory: either they are too conservative and reject valid programs, or fail to detect some shape mismatch errors.

This paper pursuits the type-based approach as it is expected to provide modular detection of tensor shape inconsistencies. Designing an appropriate type system and a type inference procedure to reason about tensor shapes is challenging because shapes are first-class objects. For example, the library function `Tensor.zeros` of OCaml-Torch [4] (which provides OCaml bindings for libtorch [20]) takes a list S of integers, and returns a new tensor whose shape is S. Thus, we have to work with *dependent types*: `Tensor.zeros` would be given the type $S : \texttt{int list} \to \{r : \texttt{tensor} \mid r.\texttt{shape} = S\}$. It is difficult to infer such dependent (refinement) types fully automatically. Yet, we wish to avoid programmers' burden of writing too many type annotations.

Another difficulty is that shape constraints can be so complex that even type checking, let alone inference, can be too costly or impossible. For instance, the reshape operation explained earlier needs the proof that the shape of the input tensor x is compatible with the given shape $S = [s_1; \ldots; s_n]$ (i.e., if the shape of x is to be $[s'_1; \ldots; s'_m]$, then $\Pi_{i=1}^{m} s'_i = \Pi_{i=1}^{n} s_i$ holds)[2]. Thus, type checking requires complex reasoning about (non-linear) integer arithmetic and lists.

Overview of Our Approach. Based on the observations above, we propose an approach that is expected to work well in practice despite the above-mentioned difficulties. Our approach can be characterized by three main features: best-effort type inference, hybrid type checking, and gradual typing [27]. We explain them using our prototype tool GraTen[3].

Best-Effort Type Inference. GraTen does not try to infer the *most general* types; it performs type/shape inference in a *best-effort* manner. Thanks to this design choice, GraTen works even if no type annotations are provided (despite that the underlying type system involves dependent types), and yet it can statically detect (not necessarily all but) some shape mismatch errors.

As an example, let us consider the program in Figure 1. The function `model` takes an integer parameter `s`, defines functions `f` and `g`, and returns a layer (which is a function that takes a tensor and returns a tensor) which composes `f`

[2] Actually, some s_i can be -1, in which case the size of the i-th dimension is unspecified.

[3] The tool is publicly available at https://doi.org/10.5281/zenodo.7590480. The source code is also publicly available at https://github.com/momohatt/graten.

```
let model s =
  let f = ... in let g = ... in
  fun x -> let y = if s = 1 then x else f x in g y
```

Fig. 2. The program from Figure 1 with small modification.

```
let model s =
  let f = ... in let g = ... in
  fun x -> let y = if s = 1 then x else f x in
           g (assert (y.shape = [10]); y)
```

Fig. 3. The program returned by GraTen given the program in Figure 2.

and g. The definitions of f and g are omitted here, but their types are assumed as below, where s in the type of f is the argument of model and the function $\mathtt{nth}(n, S)$ returns the n-th element of the list S (the index starts with 0).

$$\mathtt{f} : x{:}\{\nu : \mathtt{tensor} \mid \mathtt{len}(\nu.\mathtt{shape}) = 1\} \rightarrow \mathtt{tensor}\,([\mathtt{nth}(0, x.\mathtt{shape})/\mathtt{s}])$$

$$\mathtt{g} : \mathtt{tensor}([10]) \rightarrow \mathtt{tensor}([1])$$

These types indicate that f takes a 1-dimensional tensor (i.e., a vector) and returns a vector whose length equals the length of the argument vector divided by s, and that g expects a vector of length 10 and returns a vector of length 1. The formal syntax of types will be introduced later in Section 2.

For the program above, GraTen's best-effort inference outputs the following type for the function model.

$$s{:}\mathtt{int} \rightarrow x{:}\,\{\nu{:}\mathtt{tensor} \mid \mathtt{len}(\nu.\mathtt{shape}) = 1 \wedge \mathtt{nth}(0, \nu.\mathtt{shape})/\mathtt{s} = 10\} \rightarrow \mathtt{tensor}([1])$$

Here, the constraint $\mathtt{nth}(0, \nu.\mathtt{shape})/\mathtt{s} = 10$ for the shape of x is necessary for this program not to raise a shape mismatch error at the application of g. The inferred type of model is used to prevent any calls to model that violate the constraint. Indeed, GraTen rejects the call on line 4 of Figure 1, where the arguments do not satisfy the constraint $\frac{\mathtt{nth}(0,\nu.\mathtt{shape})}{\mathtt{s}} = 10$. As in this example, our approach can statically detect shape mismatches when enough type information has been obtained from the best-effort type inference or user-provided type annotations.

Hybrid Type Checking. Another main feature of our approach is hybrid type checking: we combine static and dynamic checking. The type checker inserts assertions to program points where the type safety is not statically guaranteed, à la Knowles and Flanagan's hybrid type checking [16]. For example, consider the program in Figure 2, which is obtained by adding a conditional branch to the one in Figure 1. The type of the then and else branch of the if expression are inferred to be $\mathtt{tensor}(\mathtt{x.shape})$ and $\mathtt{tensor}([\frac{\mathtt{nth}(0,\mathtt{x.shape})}{\mathtt{s}}])$, respectively. In this case, the type of y is simply inferred to be tensor without any information about its shape, and the inferred type for model is as follows.

$$s{:}\mathtt{int} \rightarrow x{:}\{\nu : \mathtt{tensor} \mid \mathtt{len}(\nu.\mathtt{shape}) = 1\} \rightarrow \mathtt{tensor}([1])$$

Thus, the best-effort inference of GraTen fails to capture the constraint $\frac{\mathtt{nth}(0,\nu.\mathtt{shape})}{s} = 10$ for x due to the imprecise type information of y. Along with

```
let model s =
  let f = ... in let g = ... in
  fun x ->
    let y = ((if s = 1 then x else f x) : tensor([nth 0 x.shape / s]))
    in g y
```

Fig. 4. The program from Figure 2 after adding type annotations.

the inferred types, GraTen outputs the program in Figure 3, which is the same as the original program except for the assertion inserted at the argument of `g`. Since the statically inferred type of `y` fails to guarantee that the application of `g` to `y` does not leads to a shape mismatch error, GraTen inserts the assertion to check the requirement dynamically.

Gradual Typing. Lastly, our approach incorporates *gradual typing* [27][4] so that the users can improve the precision of inferred types by adding type annotations. For example, let us consider the program in Figure 4, which is obtained from the one in Figure 2 by adding a type annotation to `y`. With this annotation, GraTen infers the same type for `model` as it did for `model` in Figure 1, and no assertions are inserted. As such, adding correct type annotations improves the type checking and decreases the number of assertions inserted.

Thanks to the best-effort inference, users need not add type annotations to everywhere in the program. They can focus on the program points where the static inference did not perform well, which is indicated by the insertion of assertions. We prove that our type system satisfies the gradual guarantee [27], which ensures that adding type annotation preserves the type-ability and the behavior of the program (with some assertions inserted) regardless of its precision, as long as the annotation does not disagree with the program.

Among the three features, the notion of hybrid type checking was first proposed by Knowles and Flanagan [16], and our gradual typing is closely related to gradual refinement types by [18], but we believe that the particular combination of three features is new. In particular, unlike the original gradual refinement types [18], we insert assertions instead of carrying around evidence terms [11] in the reduction to guarantee type safety.

The contributions are summarized as follows. (i) The formalization of a type system that combines hybrid type checking and gradual typing. We define our type system as the type-based transformation relation from source programs to programs with run-time assertion checks. We prove the soundness of our type system as well (Section 2). (ii) A proof that our system satisfies the gradual guarantee [27] (Section 3). (iii) Implementation of a best-effort type inference

[4] Usually, gradual typing introduces new syntax for gradual types and makes a distinction between static types and gradual types. However, our type system does not have such distinction; it only uses the standard refinement types. As we see later, we extend the standard refinement type system with cast (assertion) insertion rules so that it can be viewed as a gradualized type system.

$$
\begin{aligned}
M\,(\text{term}) &::= c \mid x \mid \lambda x{:}\tau.M \mid M\,x \mid (M : \tau) \mid \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2 \\
&\mid\quad \mathtt{fix}(f{:}(x{:}\tau_1 \to \tau_2), x, M) \mid \mathtt{if}\ x\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 \\
\tau\,(\text{type}) &::= \{x : B \mid \varphi\} \mid x{:}\tau_1 \to \tau_2 \\
\Gamma\,(\text{type env.}) &::= \varnothing \mid \Gamma, x : \tau \qquad \Delta\,(\text{base type env.}) ::= \varnothing \mid \Delta, x : B
\end{aligned}
$$

Fig. 5. Syntax of the source language, the types and the type environments.

on a prototype system GraTen inference (Section 4). (iv) Experimental evaluation of GraTen using the examples of deep learning programs bundled in the OCaml-Torch library. We confirm that GraTen can statically type-check the programs effectively with a reasonable amount of type annotations (Section 5).

2 A Gradually-Typed Language with Refinement Types

In this section, we formalize our type system and the translation to insert assertions. We first introduce the source and target languages of the translation in Sections 2.1 and 2.2. We then formalize the type system and the translation and prove their soundness in Section 2.3. The gradual guarantee is discussed later in Section 3.

2.1 Source Language

We consider a call-by-value functional language, whose syntax is given in Figure 5. Throughout this paper, n, c, and x respectively denote integers, constants (including integers and primitive functions) and variables. The base types B and refinement predicates φ are explained later.

Type annotations can be added to the function arguments $\lambda x{:}\tau.M$, recursive functions $\mathtt{fix}(f{:}(x{:}\tau_1 \to \tau_2), x, M)$ and to arbitrary expressions by $(M : \tau)$. In the implementation of GraTen, users may omit the type annotations in lambda expressions and recursive functions as the best-effort type inference tries to complete them.

The argument of a function application and the branching condition of an if-expression are restricted to variables for the sake of simplicity of typing rules. Note that this restriction does not lose generality, as a general function application $M_1\,M_2$ can be normalized to $\mathtt{let}\ f = M_1\ \mathtt{in}\ \mathtt{let}\ x = M_2\ \mathtt{in}\ f\,x$.

Types are defined following the standard definition of refinement types. Intuitively, the type $\{x{:}B \mid \varphi\}$ describes a value x of type B such that φ holds. For example, $\{x{:}\mathtt{int} \mid x \geq 0\}$ is the type of non-negative ints. We may omit the refinement predicates when they are $\mathtt{true}$. For example, we may write $\{x{:}\mathtt{int} \mid \mathtt{true}\}$ as $\mathtt{int}$.

The language presented so far is general; in GraTen it is instantiated to a language for tensor programs by defining the base types and refinement predicates as in Figure 6, and assuming that primitive operations on tensors are included in the set of constants ranged over c. The refinement predicates, shapes

$$\begin{array}{rl}
B\,(\text{base type}) ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{int list} \mid \texttt{tensor} \\
\varphi\,(\text{predicate}) ::= & \texttt{true} \mid \texttt{false} \mid s_1 = s_2 \mid S_1 = S_2 \mid x \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \\
\mid & \texttt{broadcastable}(S_1, S_2) \mid \texttt{reshapeable}(S_1, S_2) \\
S\,(\text{shape}) ::= & [s_1; \dots; s_n] \mid x \mid x.\texttt{shape} \mid \texttt{cons}(s, S) \mid \texttt{append}(S_1, S_2) \mid \texttt{tail}(S) \\
\mid & \texttt{init}(S) \mid \texttt{insertAt}(s_1, s_2, S) \mid \texttt{dropAt}(s, S) \mid \texttt{swap}(s_1, s_2, S) \\
\mid & \texttt{reshape}(S_1, S_2) \mid \texttt{broadcast}(S_1, S_2) \mid \texttt{matmul}(S_1, S_2) \\
s\,(\text{size}) ::= & n \mid x \mid -s \mid s_1 + s_2 \mid s_1 \times s_2 \mid \dfrac{s_1}{s_2} \mid \texttt{head}(S) \mid \texttt{last}(S) \\
\mid & \texttt{len}(S) \mid \texttt{nth}(s, S) \mid \texttt{prod}(S)
\end{array}$$

Fig. 6. Syntax of base types B and predicates φ in GraTen.

$$\begin{array}{rl}
v\,(\text{value}) ::= & c \mid x \mid [v_1, \dots, v_n] \mid \lambda x^{\tau}.N \mid \texttt{fix}(f^{\tau}, x, N) \\
N\,(\text{cast term}) ::= & v \mid \texttt{if } v \texttt{ then } N_1 \texttt{ else } N_2 \mid N\,v \mid \texttt{let } x^{\tau} = N_1 \texttt{ in } N_2 \mid \textbf{assert}(\varphi); N
\end{array}$$

Fig. 7. Syntax of the target language.

and sizes are expressions of type `bool`, `int list` and `int` respectively. The supported predicates are those described by quantifier-free formulas of first-order logic. As shown in the definition, they may use some built-in predicates and functions over integer lists such as `append` and primitives on integer arithmetic in order to express common tensor operations. We implicitly assume that the refinement predicates are well formed (as defined in the full version [13]).

2.2 Target Language

As explained in Section 1, we insert run-time checks into places where type-safety cannot be statically guaranteed. Figure 7 shows the syntax of programs obtained by the insertion of assertions. A main difference from the source language is the addition of assertion $\textbf{assert}(\varphi); N$, which is used to implement the run-time checks. Like Flanagan's hybrid type system [16] (and unlike the blame calculus [32]), we guarantee the safety of target programs by assertions. Compared with the blame calculus, this method is expected to be easier to implement since most of the modern programming languages are equipped with assertions, and more efficient in that it avoids the accumulation of dynamic casts at runtime. This implementation of the dynamic cast is possible since our system is only "gradualized" at the predicate level of the refinement type and the underlying simple type is static.

Another difference is that the binders in `let` expressions are annotated with their type. This is required when defining the precision relation over the cast terms in Section 3.

The substitution and the reduction rules of the cast terms are presented in Figure 8. The evaluation of primitive function $\texttt{ev}(c, v)$ is defined to be the return value of the primitive function c applied to an argument v if v meets the

$\boxed{[v/x]N}$

$$[v/x](\mathbf{assert}(\varphi); N) = \mathbf{assert}([v/x]\varphi); [v/x]N$$
$$[v/x](\lambda y^{\tau}.N) = \lambda y^{[v/x]\tau}.[v/x]N$$

(Variables are assumed to be alpha-renamed so that variables at different scopes do not collide)

$\boxed{N_1 \longrightarrow N_2}$

$$\mathbf{assert}(\mathtt{true}); N \longrightarrow N$$
$$\mathbf{assert}(\mathtt{false}); N \longrightarrow \mathtt{error}$$
$$c\, v \longrightarrow \mathtt{ev}(c, v)$$
$$(\lambda x^{\tau}.N_1)\, v \longrightarrow [v/x]N_1$$

Fig. 8. Selected rules of substitution and reduction of the target language (the full definition is given in the full version [13]).

$$\Gamma; \varphi \vdash c : ty(c) \;\text{(CT-Con)} \qquad \frac{\Gamma(x) = y{:}\tau_1 \to \tau_2}{\Gamma; \varphi \vdash x : \Gamma(x)} \;\text{(CT-VF)} \qquad \frac{\Gamma(x) = \{y : B \mid \varphi'\}}{\Gamma; \varphi \vdash x : \{y : B \mid y = x\}} \;\text{(CT-VB)}$$

$$\frac{\Gamma, x : \tau_1; \varphi \vdash N : \tau_2}{\Gamma; \varphi \vdash \lambda x^{\tau_1}.N : x{:}\tau_1 \to \tau_2} \;\text{(CT-Lam)} \qquad \frac{\Gamma; \varphi \vdash N : x{:}\tau_1 \to \tau_2 \qquad \Gamma; \varphi \vdash v : \tau_1}{\Gamma; \varphi \vdash N\, v : [v/x]\tau_2} \;\text{(CT-App)}$$

$$\frac{\Gamma, f : (x{:}\tau_1 \to \tau_2), x : \tau_1; \varphi \vdash N : \tau_2}{\Gamma; \varphi \vdash \mathtt{fix}(f^{x:\tau_1 \to \tau_2}, x, N) : x{:}\tau_1 \to \tau_2} \;\text{(CT-Fix)} \qquad \frac{\Gamma; \varphi \wedge \varphi' \vdash N : \tau}{\Gamma; \varphi \vdash \mathbf{assert}(\varphi'); N : \tau} \;\text{(CT-Ass)}$$

$$\frac{\Gamma; \varphi \vdash v : \{x : \mathtt{bool} \mid \varphi'\} \qquad \Gamma; \varphi \wedge v \vdash N_1 : \tau \qquad \Gamma; \varphi \wedge \neg v \vdash N_2 : \tau}{\Gamma; \varphi \vdash \mathtt{if}\ v\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : \tau} \;\text{(CT-If)}$$

$$\frac{\Gamma; \varphi \vdash N_1 : \tau_1 \qquad \Gamma, x : \tau_1; \varphi \vdash N_2 : \tau}{\Gamma; \varphi \vdash \mathtt{let}\ x^{\tau_1} = N_1\ \mathtt{in}\ N_2 : \tau} \;\text{(CT-Let)} \qquad \frac{\Gamma; \varphi \vdash N : \tau' \qquad \Gamma; \varphi \vdash \tau' <: \tau}{\Gamma; \varphi \vdash N : \tau} \;\text{(CT-Sub)}$$

Fig. 9. Typing rules for the cast terms $\Gamma; \varphi \vdash N : \tau$.

constraint of the argument of c, and otherwise undefined. We denote $N \Uparrow$ if there exists an infinite reduction sequence from N.

The substitution for cast terms is defined in the standard manner, except that the implicitly-annotated type information and the predicate in the assertion need to be updated as well. As can be seen in the definition of the cast term reduction, these implicitly-annotated types are only required for the sake of formalization and ignored at runtime.

We also introduce the type derivation rules for the cast terms $\Gamma; \varphi \vdash N : \tau$ in Figure 9. This relation is used in the discussion of the soundness of the type system later in Section 2.3. The quadruple relation $\Gamma; \varphi \vdash N : \tau$ denotes that a cast term N has type τ under a type environment Γ and a logical context φ. The logical context φ holds the information of logically valid predicates at respective program points. New predicates are added at the `then` branch and the `else` branch of (CT-If), and the post-assertion cast term in (CT-Ass). The subsumption is allowed in (CT-Sub) by the subtyping relation $\Gamma; \varphi \vdash \tau_1 <: \tau_2$ (Figure 10), which is defined in a standard manner.

$\boxed{\Phi(\Gamma), \mathtt{BT}(\Gamma)}$

$$\begin{aligned}
\Phi(\varnothing) &= \mathtt{true}\\
\Phi(\Gamma, x : \{y : B \mid \varphi\}) &= \Phi(\Gamma) \wedge [x/y]\varphi\\
\Phi(\Gamma, x : (y{:}\tau_1 \to \tau_2)) &= \Phi(\Gamma)\\
\mathtt{BT}(\varnothing) &= \varnothing\\
\mathtt{BT}(\Gamma, x : \{x : B \mid \varphi\}) &= \mathtt{BT}(\Gamma), x : B\\
\mathtt{BT}(\Gamma, x : (y{:}\tau_1 \to \tau_2)) &= \mathtt{BT}(\Gamma)
\end{aligned}$$

$\boxed{\Gamma; \varphi \vdash \tau_1 <: \tau_2}$

$$\frac{\models \forall \mathtt{BT}(\Gamma), x{:}B.\Phi(\Gamma) \wedge \varphi \wedge \varphi_1 \Rightarrow \varphi_2}{\Gamma; \varphi \vdash \{x : B \mid \varphi_1\} <: \{x : B \mid \varphi_2\}} \text{(Sub-Base)}$$

$$\frac{\Gamma; \varphi \vdash \tau_3 <: \tau_1 \qquad \Gamma, x : \tau_3; \varphi \vdash \tau_2 <: \tau_4}{\Gamma; \varphi \vdash x{:}\tau_1 \to \tau_2 <: x{:}\tau_3 \to \tau_4} \text{(Sub-Fun)}$$

Fig. 10. Subtyping rules.

$$\Gamma; \varphi \vdash c \rightsquigarrow c : ty(c) \text{ (CI-Const)}$$

$$\frac{\Gamma(x) = y{:}\tau_1 \to \tau_2}{\Gamma; \varphi \vdash x \rightsquigarrow x : \Gamma(x)} \text{(CI-Var-Fun)}$$

$$\frac{\Gamma(x) = \{y : B \mid \varphi'\}}{\Gamma; \varphi \vdash x \rightsquigarrow x : \{y : B \mid y = x\}} \text{(CI-Var-Base)}$$

$$\frac{\Gamma, x : \tau_1; \varphi \vdash M \rightsquigarrow N : \tau_2}{\Gamma; \varphi \vdash \lambda x{:}\tau_1.M \rightsquigarrow \lambda x^{\tau_1}.N : x{:}\tau_1 \to \tau_2} \text{(CI-Lam)}$$

$$\frac{\Gamma; \varphi \vdash M_1 \rightsquigarrow N_1 : y{:}\tau_1 \to \tau_2 \qquad \Gamma(x) = \tau_3 \qquad \Gamma; \varphi \vdash \tau_3 \lesssim \tau_1 \rightsquigarrow N_2}{\Gamma; \varphi \vdash M_1\, x \rightsquigarrow (\mathtt{let}\ x^{\tau_1} = N_2\, x\ \mathtt{in}\ N_1\, x) : [x/y]\tau_2} \text{(CI-App)}$$

$$\frac{\Gamma, f : (x{:}\tau_1 \to \tau_2), x : \tau_1; \varphi \vdash M \rightsquigarrow N : \tau_2}{\Gamma; \varphi \vdash \mathtt{fix}(f{:}(x{:}\tau_1 \to \tau_2), x, M) \rightsquigarrow \mathtt{fix}(f^{x:\tau_1 \to \tau_2}, x, N) : x{:}\tau_1 \to \tau_2} \text{(CI-Fix)}$$

$$\frac{\Gamma; \varphi \vdash M_1 \rightsquigarrow N_1 : \tau_1 \qquad \Gamma, x : \tau_1; \varphi \vdash M_2 \rightsquigarrow N_2 : \tau \qquad \mathtt{BT}(\Gamma) \vdash_{\mathtt{wf}} \tau}{\Gamma; \varphi \vdash (\mathtt{let}\ x = M_1\ \mathtt{in}\ M_2) \rightsquigarrow (\mathtt{let}\ x^{\tau_1} = N_1\ \mathtt{in}\ N_2) : \tau} \text{(CI-Let)}$$

$$\frac{\Gamma; \varphi \vdash v : \{x : \mathtt{bool} \mid \varphi'\} \qquad \Gamma; \varphi \wedge v \vdash M_1 \rightsquigarrow N_1 : \tau \qquad \Gamma; \varphi \wedge \neg v \vdash M_2 \rightsquigarrow N_2 : \tau}{\Gamma; \varphi \vdash \mathtt{if}\ v\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 \rightsquigarrow \mathtt{if}\ v\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : \tau} \text{(CI-If)}$$

$$\frac{\Gamma; \varphi \vdash M \rightsquigarrow N : \tau}{\Gamma; \varphi \vdash (M : \tau) \rightsquigarrow N : \tau} \text{(CI-Annot)}$$

$$\frac{\Gamma; \varphi \vdash M_1 \rightsquigarrow N_1 : \tau_1 \qquad \Gamma; \varphi \vdash \tau_1 \lesssim \tau \rightsquigarrow N_2}{\Gamma; \varphi \vdash M_1 \rightsquigarrow \mathtt{let}\ x^{\tau_1} = N_1\ \mathtt{in}\ N_2\, x : \tau} \text{(CI-Sub)}$$

Fig. 11. Type derivation rules for the source language $\Gamma; \varphi \vdash M \rightsquigarrow N : \tau$.

2.3 Typing Rules

Inserting Assertions Next, we discuss the typing rules for the source language and the assertion insertion into it. Figure 11 defines the type judgement and cast insertion relation. The intuition of 5-ary relation $\Gamma; \varphi \vdash M \rightsquigarrow N : \tau$ is: under a type environment Γ and a logical context φ, a term M translates to a cast term N and has type τ. If we ignore the part "$\rightsquigarrow N$" and replace the gradual subtyping relation $\lesssim$ with the standard subtyping relation on refinement types (Figure 10), our type system is a standard refinement type system. Thus, the main novelty in the rules in Figure 11 lies in the use of the *consistent subtyping relation* $\Gamma; \varphi \vdash \tau_1 \lesssim \tau_2 \rightsquigarrow N$, which is explained below.

The consistent subtyping relation $\Gamma; \varphi \vdash \tau_1 \lesssim \tau_2 \rightsquigarrow N$ (Figure 12)[5] is used in the cast insertion relation to guarantee that there exists a value that has both of the types τ_1 and τ_2 under Γ and φ, and to produce an assertion term N that checks at runtime if a value that is statically known to be of type τ_1 can be used as a value of type τ_2.

The rule for the base case (Cast-Base) checks if there exists a value, and an assignment of the values to the variables in the type environment, that satisfies both τ_1 and τ_2. This intuitively holds if τ_1 is castable to τ_2 for some runtime values. The rule also produces a lambda function that implements the cast with an assertion. It is defined in such a way that φ_2 can always be used as the content of the assertion φ', but `true` can also be used for φ' if φ_1 implies φ_2. Note that we cannot use φ_2 as the content of the assertion in the definition, or otherwise Proposition 1 does not hold.

The rule for the function types (Cast-Fun) recursively checks the castability of the argument types and the return types and combines the assertion terms for them. Notice how the subsumption for the return types τ_2 and τ_4 has the meet of two argument types $\tau_1 \sqcap \tau_3$ in the type environment. The meet of two types (Figure 12) is defined as a conjunction of the refinement predicates[6].

The consistent subtyping relation can be seen as a gradualization of the subtyping relation $\Gamma; \varphi \vdash \tau_1 <: \tau_2$ (Figure 10). In fact, when a type τ_1 is a subtype of another type τ_2, it is possible that the assertion term generated by casting τ_1 to τ_2 only contains assertions that always succeed, which can be erased by some optimization. The following proposition states this fact. Note that this corresponds to the blame-subtyping theorem, one of the criteria for gradual typing presented in [27].

Proposition 1. *$\Gamma; \varphi \vdash \tau_1 <: \tau_2$ implies $\Gamma; \varphi \vdash \tau_1 \lesssim \tau_2 \rightsquigarrow N$ for some N where all the assertions in N are of the form* **assert**(`true`)*; N'.*

[5] This can be understood as the refinement-type version of the differential subtyping in [23], although in the implementation we do not calculate the "difference" between φ_1 and φ_2 for φ' in the assertion unless φ_1 implies φ_2 (and thus φ' can be `true`).

[6] Although the meet of two function types is defined, it does not make any difference in the definition of consistent subtyping relation since function types in the type environment is not used.

$$\{x : B \mid \varphi_1\} \sqcap \{x : B \mid \varphi_2\} = \{x : B \mid \varphi_1 \wedge \varphi_2\}$$
$$(x{:}\tau_1 \to \tau_2) \sqcap (x{:}\tau_3 \to \tau_4) = x{:}(\tau_1 \sqcap \tau_3) \to (\tau_2 \sqcap \tau_4)$$

$$\frac{\models \exists \mathtt{BT}(\Gamma), x{:}B.\Phi(\Gamma) \wedge \varphi \wedge \varphi_1 \wedge \varphi_2 \qquad \models \forall \mathtt{BT}(\Gamma), x{:}B.\Phi(\Gamma) \wedge \varphi \wedge \varphi_1 \Rightarrow (\varphi' \Leftrightarrow \varphi_2)}{\Gamma; \varphi \vdash \{x : B \mid \varphi_1\} \lesssim \{x : B \mid \varphi_2\} \rightsquigarrow \lambda x^{\{x:B|\varphi_1\}}.\mathbf{assert}(\varphi'); x} \;(\textsc{Cast-Base})$$

$$\frac{\Gamma; \varphi \vdash \tau_3 \lesssim \tau_1 \rightsquigarrow N_1 \qquad \Gamma, x : \tau_1 \sqcap \tau_3; \varphi \vdash \tau_2 \lesssim \tau_4 \rightsquigarrow N_2}{\begin{array}{c}\Gamma; \varphi \vdash x{:}\tau_1 \to \tau_2 \lesssim x{:}\tau_3 \to \tau_4 \rightsquigarrow \\ \lambda f^{x:\tau_1 \to \tau_2}.\lambda x^{\tau_3}.(\mathtt{let}\ y^{\tau_1 \sqcap \tau_3} = N_1\, x\ \mathtt{in}\ \mathtt{let}\ z^{\tau_2} = f\, y\ \mathtt{in}\ N_2\, z)\end{array}} \;(\textsc{Cast-Fun})$$

Fig. 12. Definition of the consistent subtyping relation $\Gamma; \varphi \vdash \tau_1 \lesssim \tau_2 \rightsquigarrow N$.

Type Safety We conclude this section with a note on the soundness of our type system. The soundness is based on the fact that if the source program is well-typed, the program after the assertion insertion is also well-typed.

The most critical part of the proof is to prove the assertion term can be assigned a function type from the pre-assertion type to the post-assertion type.

Lemma 1. *$\Gamma; \varphi \vdash \tau_1 \lesssim \tau_2 \rightsquigarrow N$ implies $\Gamma; \varphi \vdash N : x{:}\tau_1 \to \tau_2$ for some variable x that does not occur in τ_2.*

The proof is found in the full version [13]. With Lemma 1, we can prove that the assertion-inserted program can be assigned the same type as that of the original program.

Lemma 2 (Assertion Insertion Preserves Types). *$\Gamma; \varphi \vdash M \rightsquigarrow N : \tau$ implies $\Gamma; \varphi \vdash N : \tau$.*

We can also prove the standard progress and preservation properties under a reasonable assumption that the types of the primitive functions are properly defined as follows (see the full version [13] for the proofs).

Assumption 1 *$\vdash c\, v : \tau$ implies $ev(c, v)$ is defined and $\vdash ev(c, v) : \tau$*

Combining Lemma 2 with the progress and preservation properties, we obtain the type safety as follows.

Theorem 1 (Type Safety). *With Assumption 1, $\varnothing; \mathtt{true} \vdash M \rightsquigarrow N : \tau$ implies $N \longrightarrow^* v$ for some v, $N \Uparrow$, or $N \longrightarrow^* \mathtt{error}$.*

The type safety property states that a well-typed program does not cause untrapped dynamic errors. The only case where a cast-inserted program causes untrapped errors is when the result of an application of a primitive function is undefined (i.e., $ev(c, v)$ is undefined). The type safety property ensures that such untrapped errors do not happen for well-typed terms as long as the $ty(c)$ is defined appropriately.

$$\boxed{\widetilde{x} \vdash \tau_1 \sqsubseteq \tau_2}$$

$$\frac{\models \forall \widetilde{y}, x.\varphi_1 \Rightarrow \varphi_2}{\widetilde{y} \vdash \{x : B \mid \varphi_1\} \sqsubseteq \{x : B \mid \varphi_2\}} \text{ (PREC-BASE)}$$

$$\frac{\widetilde{y} \vdash \tau_1 \sqsubseteq \tau_3 \qquad \widetilde{y}, x \vdash \tau_2 \sqsubseteq \tau_4}{\widetilde{y} \vdash x{:}\tau_1 \to \tau_2 \sqsubseteq x{:}\tau_3 \to \tau_4} \text{ (PREC-FUN)}$$

$$\boxed{\Gamma_1 \sqsubseteq \Gamma_2}$$

$$\varnothing \sqsubseteq \varnothing$$

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2 \qquad dom(\Gamma_1) \vdash \tau_1 \sqsubseteq \tau_2}{\Gamma_1, x : \tau_1 \sqsubseteq \Gamma_2, x : \tau_2}$$

Fig. 13. Precision relation of types and type environments.

3 Gradual Guarantee

In a standard gradual type system, programs are compared by their *precision*, or the amount of information contained in the type annotations. This notion is used to define the gradual guarantee [27], which is the core property of gradual typing. The gradual guarantee comes in two parts. The first one is called *static gradual guarantee*, which states that decreasing the precision of type annotation from a well-typed program still preserves the typeability of the program at a less precise type. The second one is called *dynamic gradual guarantee*, which claims that a less precise program behaves the same as the more precise one with fewer assertion errors.

Below we first define the precision for the language introduced in Section 2. We then show that our type system satisfies the gradual guarantee.

Precision. Figure 13 defines the precision relation $\widetilde{x} \vdash \tau_1 \sqsubseteq \tau_2$ on types by using the logical implication between the refinement predicates. The sequence of variables $\widetilde{x}$ keeps the variables that may appear in the refinement predicates. For example, the following is an example of the type precision relation for the base type.

$$\vdash \{x : \texttt{tensor} \mid x.\texttt{shape} = [3]\} \sqsubseteq \{x : \texttt{tensor} \mid \texttt{len}(x.\texttt{shape}) = 1\}$$

Note that in the rule (PREC-FUN), the precision of the argument type and the return type are compared independently; the type information on x is not used in the comparison of the return types. This is in contrast with the rule (SUB-FUN) in Figure 10 for subtyping. Figure 13 also extends the relation to $\Gamma \sqsubseteq \Gamma'$ on type environments. The precision relation is also extended to the relation $\widetilde{x} \vdash M \sqsubseteq M'$ on terms, by the rules in Figure 14. Here, $\widetilde{x}$ is the sequence of variables in scope. Finally, we define the precision relation of the cast terms in Figure 14. Unlike the term precision relation (Figure 14), the precision relation $\Gamma; \varphi \vdash N_1 \sqsubseteq N_2$ on cast terms requires the type environment Γ and the logical context φ in the judgement, and the refinement extraction from the type environment $\Phi(\Gamma)$ is used in the rule (PC-ASSERT). We also assume the following property on the evaluation of the primitive functions.

Assumption 2 *If $ev(c, v_2)$ and $ev(c, v_1)$ are both defined, then $v_1 \sqsubseteq v_2$ implies $ev(c, v_1) \sqsubseteq ev(c, v_2)$*

$\boxed{\widetilde{x} \vdash M_1 \sqsubseteq M_2}$

$$\frac{\widetilde{y} \vdash \tau_1 \sqsubseteq \tau_2 \qquad \widetilde{y}, x \vdash M_1 \sqsubseteq M_2}{\widetilde{y} \vdash \lambda x{:}\tau_1.M_1 \sqsubseteq \lambda x{:}\tau_2.M_2} \text{(PM-Lam)} \qquad \frac{\widetilde{y} \vdash M_1 \sqsubseteq M_2 \qquad \widetilde{y} \vdash \tau_1 \sqsubseteq \tau_2}{\widetilde{y} \vdash (M_1 : \tau_1) \sqsubseteq (M_2 : \tau_2)} \text{(PM-Annot)}$$

$\boxed{\Gamma; \varphi \vdash N_1 \sqsubseteq N_2}$

$$\frac{\forall \mathtt{BT}(\Gamma).\Phi(\Gamma) \wedge \varphi \wedge \varphi_1 \Rightarrow \varphi_2 \qquad \Gamma; \varphi \wedge \varphi_1 \vdash N_1 \sqsubseteq N_2}{\Gamma; \varphi \vdash \mathbf{assert}(\varphi_1); N_1 \sqsubseteq \mathbf{assert}(\varphi_2); N_2} \text{(PC-Assert)}$$

Fig. 14. Selected rules for the precision relation on terms and cast terms (the full definition is found in the full version [13]).

Intuitively, the precision of cast terms are designed in such a way that, when $\varnothing; \mathtt{true} \vdash N_1 \sqsubseteq N_2$ holds, the assertions in N_1 is more strict than that of N_2, and therefore the dynamic checks in N_1 is more likely to fail than in N_2. The following two propositions state this intuition (the proofs are found in the full version [13]).

Proposition 2. *Suppose $\varnothing; \mathtt{true} \vdash N_1 : \tau$ and $\varnothing; \mathtt{true} \vdash N_2 : \tau'$. Then, $\varnothing; \mathtt{true} \vdash N_1 \sqsubseteq N_2$ and $N_1 \longrightarrow N_1'$ imply $N_2 \longrightarrow N_2'$ and $\varnothing; \mathtt{true} \vdash N_1' \sqsubseteq N_2'$ for some N_2'.*

Proposition 3. *Suppose $\varnothing; \mathtt{true} \vdash N_1 : \tau$ and $\varnothing; \mathtt{true} \vdash N_2 : \tau'$. Then, $\varnothing; \mathtt{true} \vdash N_1 \sqsubseteq N_2$ and $N_2 \longrightarrow N_2'$ imply either of the following.*

- *$N_1 \longrightarrow N_1'$ and $N_1' \sqsubseteq N_2'$ for some N_1'*
- *$N_1 \longrightarrow \mathtt{error}$*

Gradual Guarantee. We show that our system satisfies the gradual guarantee [27]. First, we prove that the consistent subtyping relation $\Gamma; \varphi \vdash \tau_1 \lesssim \tau_2 \rightsquigarrow N$ is upper-closed with respect to the precision relation $\widetilde{x} \vdash \tau_1 \sqsubseteq \tau_3$ on types.

Lemma 3. *$\Gamma; \varphi \vdash \tau_1 \lesssim \tau_2 \rightsquigarrow N_1$, $dom(\Gamma) \vdash \tau_1 \sqsubseteq \tau_3$, $dom(\Gamma) \vdash \tau_2 \sqsubseteq \tau_4$ and $\Gamma \sqsubseteq \Gamma'$ implies $\Gamma'; \varphi \vdash \tau_3 \lesssim \tau_4 \rightsquigarrow N_2$ for some N_2.*

We can further prove that the cast term N_2 in the statement of Lemma 3 is less precise than the original cast term N_1 as follows.

Lemma 4. *Suppose $\Gamma \sqsubseteq \Gamma'$, $dom(\Gamma) \vdash \tau_1 \sqsubseteq \tau_1'$ and $dom(\Gamma) \vdash \tau_2 \sqsubseteq \tau_2'$. Then, $\Gamma; \varphi \vdash \tau_1 \lesssim \tau_2 \rightsquigarrow N$ and $\Gamma'; \varphi \vdash \tau_1' \lesssim \tau_2' \rightsquigarrow N'$ implies $\Gamma; \varphi \vdash N \sqsubseteq N'$.*

Using the above properties, we can prove the following lemma which constitutes the core part of the proof of the gradual guarantee.

Lemma 5. *$\Gamma \sqsubseteq \Gamma'$, $dom(\Gamma) \vdash M \sqsubseteq M'$ and $\Gamma; \varphi \vdash M \rightsquigarrow N : \tau$ imply $\Gamma'; \varphi \vdash M' \rightsquigarrow N' : \tau'$, $\Gamma; \varphi \vdash N \sqsubseteq N'$ and $dom(\Gamma) \vdash \tau \sqsubseteq \tau'$ for some N' and τ'.*

Finally, we can show the static and dynamic gradual guarantee as follows.

Theorem 2 (Static gradual guarantee). *$\varnothing \vdash M_1 \sqsubseteq M_2$ and $\vdash M_1 : \tau_1$ imply $\vdash M_2 : \tau_2$ and $\varnothing \vdash \tau_1 \sqsubseteq \tau_2$ for some τ_2.*

Proof. This follows immediately from Lemma 5. $\square$

Theorem 3 (Dynamic gradual guarantee). *Suppose $\varnothing \vdash M_1 \sqsubseteq M_2$ and $\vdash M_1 \rightsquigarrow N_1 : \tau_1$. Then, there exist N_2 and τ_2 that satisfy all of the following.*

- *$\vdash M_2 \rightsquigarrow N_2 : \tau_2$.*
- *$N_1 \longrightarrow^* v_1$ implies $N_2 \longrightarrow^* v_2$ and $v_1 \sqsubseteq v_2$ for some v_2.*
- *$N_1 \Uparrow$ implies $N_2 \Uparrow$.*
- *$N_2 \longrightarrow^* v_2$ implies $N_1 \longrightarrow^* v_1$ and $v_1 \sqsubseteq v_2$ for some v_1, or $N_1 \longrightarrow^*$ `error`.*
- *$N_2 \Uparrow$ implies $N_1 \Uparrow$ or $N_1 \longrightarrow^*$ `error`.*

Proof. By Lemma 5, $\vdash M_2 \rightsquigarrow N_2 : \tau_2$ holds for some N_2 and τ_2 where $\vdash N_1 \sqsubseteq N_2$ and $\vdash \tau_1 \sqsubseteq \tau_2$. Also, from Lemma 2, we obtain $\vdash N_1 : \tau_1$ and $\vdash N_2 : \tau_2$. Using Proposition 2, $N_1 \longrightarrow^* v_1$ for some v_1 implies $N_2 \longrightarrow^* v_2$ for some v_2 such that $v_1 \sqsubseteq v_2$. Also, $N_1 \longrightarrow^\infty$ implies $N_2 \longrightarrow^\infty$. Using Proposition 3, $N_2 \longrightarrow^* v_2$ for some v_2 implies $N_1 \longrightarrow^* v_1$ for some v_1 such that $v_1 \sqsubseteq v_2$, or $N_1 \longrightarrow^*$ `error`. Also, $N_2 \longrightarrow^\infty$ implies $N_1 \longrightarrow^\infty$ or $N_1 \longrightarrow^*$ `error`. $\square$

4 Best-Effort Type Inference

Thanks to our combination of gradual typing and hybrid checking described in the previous sections, a type inference procedure need not necessarily output the most precise types. It is allowed to perform type inference only in a *best-effort* manner, and the results in the previous sections do not depend on the particular design of the type inference procedure. Nevertheless, it is desirable for the procedure to infer reasonably good types. In this section, we report a specific design of the type inference procedure, which we have implemented in our prototype system GraTen; as reported in the Section 5, our procedure works reasonably well for actual deep learning programs.

4.1 Overview of Type Inference and Checking in GraTen

The type checking in GraTen consists of the following three phases: (1) simple type inference, (2) best-effort refinement type inference, and (3) consistent subtyping checking and assertion insertion.

In the first phase, GraTen performs the simple type inference using the standard Hindley-Milner algorithm and annotates the AST with the inferred simple types of each node.

In the second phase, GraTen first collects all the consistent subtyping constraints of the form $\Gamma; \varphi \vdash \tau_1 \lesssim \tau_2 \rightsquigarrow N$ from the source program. When it encounters AST nodes whose refinement type cannot be constructed directly, GraTen generates template refinement types using the simple types inferred in the previous phase. Template refinement types may contain variables for undetermined predicates (referred to as *predicate variables*).

Using the collected constraints, GraTen then tries to find a solution for all of the predicate variables with its hand-made constraint solver. The constraint solving takes place on every `let` binding to allow let-polymorphism on shapes. We discuss the detail of the implementation of the solver in the next subsection, but at a high level, the solver tries to find such a solution that:

- only general types are inferred, as otherwise it could result in rejecting well-typed programs.
- $\Gamma; \varphi \vdash \tau_1 <: \tau_2$ holds for as many constraints $\Gamma; \varphi \vdash \tau_1 \lesssim \tau_2 \rightsquigarrow N$ as possible. This is to make the cast term N consist of trivial assertions (which can statically be discharged to avoid run-time overheads; recall Proposition 1).

Given that the subtyping constraints can be expressed in the form of constrained Horn clauses (CHC) and not all the subtyping constraints need to hold, the problem above is essentially a CHC solving problem with weak constraints and maximality [22] where the optimization objective of the problem is defined by pointwise logical comparison of the solutions.

The constraint solver of GraTen does not always find a solution for all predicate variables. In such cases, GraTen assigns `true` to the undetermined predicate variables; that way, they will at least not invalidate the consistent subtyping constraints.

Note that GraTen does not take into account the consistent subtyping $\Gamma; \varphi \vdash \tau_1 \lesssim \tau_2 \rightsquigarrow N$ itself when trying to find a solution, as we expect that it would be rare for a consistent subtyping $\Gamma; \varphi \vdash \tau_1 \lesssim \tau_2 \rightsquigarrow N$ to hold when the subtyping relation $\Gamma; \varphi \vdash \tau_1 <: \tau_2$ does not hold. GraTen therefore defers the check of consistent subtyping constraints to the next phase.

In the third phase, GraTen checks the validity of consistent subtyping constraints using the inference results for the predicate variables from the previous phase. GraTen first attempts to simplify and verify the constraints by a hand-made solver, but it falls back on using z3 [5] with timeouts if it does not work. Simultaneously, it also generates the assertion terms and inserts them into the source program.

4.2 Heuristics of Best-Effort Type Inference

To solve the subtyping constraints explained above, we have implemented a hand-made constraint solver. GraTen does not use off-the-shelf SMT or CHC solvers such as Z3 [5], since the refinement predicates in GraTen often use complicated predicates on integer lists, for which standard SMT/CHC solvers cannot find a solution in a reasonable time. Also, while GraTen should infer general types (so as not to reject well-typed programs), those generic solvers are not biased towards generality and return any (non-general) solution that satisfies the constraints. This subsection describes the heuristics used in GraTen for constraint solving.

The preparation for the inference is already started when GraTen generates the template refinement types during the constraint collection. For each predicate variable generated, GraTen attaches the set of program variables it depends on, which is calculated from the type environment. This is used in the constraint solving later to avoid assigning irrelevant predicates to the predicate variables. We denote predicate variables as $p_{\widetilde{x}}(\widetilde{y})$, where $\widetilde{x}$ denotes the set of program variables it depends on and $\widetilde{y}$ denotes the parameters of the predicate variable.

After collecting the constraints, GraTen decomposes the subtyping constraints to constrained Horn clauses of the form $\widetilde{\varphi_1} \wedge \widetilde{\varphi_2} \Rightarrow \widetilde{\varphi_3}$ following the definition of the subtyping relation (Figure 10). The notation $\widetilde{\varphi}$ denotes a set of predicates, logically interpreted as the conjunction of the predicates. The first, second, and third set of predicates in the clause respectively corresponds to the predicates from the context $\Phi(\Gamma) \wedge \varphi$, the refinement of the type on the left φ_1, and that of the type on the right φ_2. We intentionally distinguish between $\widetilde{\varphi_1}$ and $\widetilde{\varphi_2}$ on the left-hand side of the clauses in describing the constraint solving algorithm. For example, let us reconsider the program in Figure 2. The subtyping constraints collected from the `if` expression of the program would be as follows, where p, q and r are the predicate variables generated for the type of `s`, `x` and the `if` expression respectively.

$$\begin{aligned}
&\Gamma; (s = 1) \vdash \{\nu{:}\texttt{tensor} \mid q_{s,\nu}(\nu)\} <: \{\nu{:}\texttt{tensor} \mid r_{s,x,\nu}(\nu)\} \\
&\Gamma; (s \neq 1) \vdash \{\nu{:}\texttt{tensor} \mid q_{s,\nu}(\nu)\} <: \{\nu{:}\texttt{tensor} \mid \texttt{len}(\nu.\texttt{shape}) = 1\} \\
&\Gamma; (s \neq 1) \vdash \texttt{tensor}([\texttt{nth}(0, x.\texttt{shape})/s]) <: \{\nu{:}\texttt{tensor} \mid r_{s,x,\nu}(\nu)\} \\
&\quad \text{where } \Gamma := [s \mapsto \{\nu{:}\texttt{int} \mid p_\nu(\nu)\}, x \mapsto \{\nu{:}\texttt{tensor} \mid q_{s,\nu}(\nu)\}]
\end{aligned}$$

These constraints are decomposed into the following clauses.

$$\begin{aligned}
\{p_s(s), q_{s,x}(x), s = 1\} \wedge \{q_{s,\nu}(\nu)\} &\Rightarrow r_{s,x,\nu}(\nu) \\
\{p_s(s), q_{s,x}(x), s \neq 1\} \wedge \{q_{s,\nu}(\nu)\} &\Rightarrow \texttt{len}(\nu.\texttt{shape}) = 1 \qquad (1) \\
\{p_s(s), q_{s,x}(x), s \neq 1\} \wedge \{\nu.\texttt{shape} = [\texttt{nth}(0, x.\texttt{shape})/s]\} &\Rightarrow r_{s,x,\nu}(\nu)
\end{aligned}$$

From the clauses obtained as above, GraTen tries to find a solution for the predicate variables using an algorithm presented in Algorithm 1.

The algorithm processes the constraints by first trying to find a solution for predicate variables that occur on the right-hand side of a clause $\widetilde{\varphi_1} \wedge \widetilde{\varphi_2} \Rightarrow \widetilde{\varphi_3}$ (Line 6-10), and then on the left-hand side of a clause (Line 11-15), and repeats it until either all of the constraints are solved or the constraints cannot be processed any further (Line 4). In Line 8 and Line 13, the set of program variables $\widetilde{x}$ of a predicate variable $p_{\widetilde{x}}$ is used to assign the predicates to the predicate variables[7].

During the iteration, the constraints need to be occasionally updated with the current solutions θ by applying the substitution θ to all the predicates in the constraints. After that, we also simplify the set of clauses (with `simplify` in Algorithm 1) by removing the predicates from the right-hand side of a clause that trivially follows from the left-hand side, and by removing clauses whose

[7] The set of program variables used in predicates is defined following the standard definition of free variables, except that the program variables used in a predicate variable $p_{\widetilde{x}}$ is defined as $\widetilde{x}$.

right-hand side is empty. For example, a clause $\{\} \wedge \{x = 1\} \Rightarrow \{x = 1\}$ is simplified to $\{\} \wedge \{x = 1\} \Rightarrow \{\}$, and then removed from the set of clauses.

To illustrate the behavior of Algorithm 1, consider applying it to the clauses (1). During the first iteration of the `while` loop (Line 4), the first `for` loop (Line 6) exits with an empty θ as r appears on the right-hand side of multiple clauses and cannot be resolved here due to the check at Line 7. In the next `for` loop (Line 11), θ is updated to:

$$[q_{s,\nu}(\nu) \mapsto (\texttt{len}(\nu.\texttt{shape}) = 1 \wedge q'_{s,\nu}(\nu))] \tag{2}$$

where $q'_{s,\nu}(\nu)$ is a fresh predicate variable, and the constraints c would be updated as follows.

$$\{p_s(s), \texttt{len}(x.\texttt{shape}) = 1, q'_{s,x}(x), s = 1\} \wedge \{\texttt{len}(\nu.\texttt{shape}) = 1 \wedge q'_{s,\nu}(\nu)\} \Rightarrow r_{s,x,\nu}(\nu)$$
$$\{p_s(s), \texttt{len}(x.\texttt{shape}) = 1, q'_{s,x}(x), s \neq 1\} \wedge \{\nu.\texttt{shape} = [\texttt{nth}(0, x.\texttt{shape})/s]\} \Rightarrow r_{s,x,\nu}(\nu)$$

The `while` loop exits after the second iteration, as no new predicate variables can be added to θ and $c = c'$ holds. Thus, we only obtain (2) from Algorithm 1. After the inference, GraTen assigns `true` to the remaining predicate variables p, q' and r.

Algorithm 1 Algorithm for calculating the solutions θ to predicate variables from constrained Horn clauses c.

```
    Input: constrained Horn clauses $c$
    Output: the mapping from predicate variables to its solution (predicates) $\theta$
 1: procedure SOLVE($c$)
 2:     Let $\theta$ be an empty substitution
 3:     $c' \leftarrow c$
 4:     while $c \neq \varnothing$ and $c \neq c'$ do
 5:         $c' \leftarrow c$
 6:         for every clause of the form $\widetilde{\varphi_1} \wedge \widetilde{\varphi_2} \Rightarrow p_{\widetilde{x}}(\widetilde{y})$ in $c$ do
 7:             if $p_{\widetilde{x}}(\widetilde{y}) \notin \widetilde{\varphi_3}'$ for any other $\widetilde{\varphi_1}' \wedge \widetilde{\varphi_2}' \Rightarrow \widetilde{\varphi_3}'$ in $c$ then
 8:                 Let $\widetilde{\varphi_2}'$ be the maximal subset of $\widetilde{\varphi_2}$ that only uses variables in $\widetilde{x}$
 9:                 $\theta \leftarrow [p_{\widetilde{x}}(\widetilde{y}) \mapsto \bigwedge \widetilde{\varphi_2}'] \circ \theta$      ▷ $\circ$ is a composition of mappings.
10:         $c \leftarrow$ simplify($\theta\, c$)      ▷ simplify($\cdot$) is described in the main text.
11:         for $\widetilde{\varphi_1} \wedge \widetilde{\varphi_2} \Rightarrow \widetilde{\varphi_3}$ in $c$ do
12:             for every predicate variable $p_{\widetilde{x}}(\widetilde{y})$ in $\widetilde{\varphi_1} \cup \widetilde{\varphi_2}$ do
13:                 Let $\widetilde{\varphi_3}'$ be the maximal subset of $\widetilde{\varphi_3}$ that only uses variables in $\widetilde{x}$
14:                 Let $q_{\widetilde{x}}(\widetilde{y})$ be a fresh predicate variable
15:                 $\theta \leftarrow [p_{\widetilde{x}}(\widetilde{y}) \mapsto (\bigwedge \widetilde{\varphi_3}') \wedge q_{\widetilde{x}}(\widetilde{y})] \circ \theta$
16:             $c \leftarrow$ simplify($\theta\, c$) ▷ Also updates the remaining items iterated by L11.
17:     return $\theta$
```

5 Experiment

This section reports on experiments to evaluate the effectiveness of our approach by running our tool GraTen for the example programs bundled in the OCaml-

Torch library [4]. We have also checked how type annotations changed the inference results.

5.1 Methods

Input and Output of GraTen GraTen takes an OCaml program and performs type checking with its best-effort type inference. If the type checking is successful, it returns the inferred types of top-level variables defined in the program, and the source program with necessary assertions inserted. Otherwise, the type checking fails with an error message.

The assertions are inserted into the output program only when they are needed. Namely, assertions are inserted into the places where the consistent subtyping $\Gamma; \varphi \vdash \tau_1 \lesssim \tau_2 \rightsquigarrow N$ is used only when $\Gamma; \varphi \vdash \tau_1 <: \tau_2$ doesn't hold (see Proposition 1).

Besides the source program, GraTen also reads the types of the library functions (including those of OCaml-Torch) from manually prepared stub files. For example, the type of `tr` (matrix transpose function) is defined as follows.

```
val tr :  x:{ v:tensor | len v.shape = 2 }
       -> tensor([nth 1 x.shape; nth 0 x.shape])
```

Note that describing the types of some higher-order OCaml-Torch functions requires the polymorphic extension, which we sketch in the full version [13]. For example, the type of `Layer.forward` is defined as follows.

$$\forall b_1{:}\texttt{bool}, b_2{:}\texttt{bool}.$$
$$(x{:}\{x{:}\texttt{tensor} \mid b_1\} \rightarrow \{y{:}\texttt{tensor} \mid b_2\}) \rightarrow x{:}\{x{:}\texttt{tensor} \mid b_1\} \rightarrow \{y{:}\texttt{tensor} \mid b_2\}$$

GraTen handles such types by instantiating the quantified parameters (b_1 and b_2 in the above case) with fresh predicate variables.

Test Cases We applied GraTen to programs under `examples/` directory of the repository of OCaml-Torch[8]. The list of programs tested is shown in Table 1. Since some programs use features of OCaml or OCaml-Torch that are not yet supported by GraTen, they were modified not to use such features without changing the structure of the neural network. Major modifications added to the target programs are listed below. Other smaller syntactic modifications can be found in the supplementary materials.

(M1) Replacing or removing type-polymorphic functions. Some functions that create loops such as `List.foldl` are replaced with recursive functions. Others such as `no_grad` are replaced with the type-instantiated versions.

(M2) Removing use of non-integer lists, especially tensor lists and layer[9] lists. As a result, two list-taking primitive functions are removed. One is `Tensor.cat`, which takes a list of tensors and returns the concatenation of them. It is

[8] https://github.com/LaurentMazare/ocaml-torch/tree/a6499811f4/examples

[9] Functions that take a tensor and return a tensor.

replaced with a variant `Tensor.cat_` which takes only two tensors. The other is `Layer.sequential`, which takes a list of layers and returns a layer that sequentially applies all the input layers.

(M3) Replacing mutable float objects with 0-dimensional tensors, as GraTen does not support reference types.

As an example of (M1) and (M2), consider the following function, which creates a list of linear layers and returns a new layer that applies all the layers in the list.

```
let f vs ~num_layers =
 List.init num_layers ~f:(fun i -> Layer.linear vs ~input_dim:(i+1) (i+2))
 |> Layer.sequential
```

The `i`-th layer in the list takes a tensor whose last dimension is size `i+1`, and returns a tensor of the same shape except that the last dimension is changed to `i+2`. By the modifications (M1) and (M2), the above function definition is replaced with:

```
let f vs ~num_layers =
 let rec loop i xs =
  if i = 0
   then Layer.id xs
   else loop (i-1) xs ~is_training |> Layer.linear vs ~input_dim:i (i+1)
 in Layer.of_fn (loop num_layers)
```

Some programs in the `examples/` directory are excluded from the test cases for the following reasons.

– `neural_transfer` uses a library function `Vgg.vgg16_layers` whose type cannot be described in GraTen; the relation between its inputs and its output tensor's shape could not be expressed in the syntax supported by GraTen.

– Programs `dqn.ml`, `dqn_atari.ml` and `dqn_pong.ml` in `reinforcement-learning` use queues which are not supported in GraTen yet.

– `env_gym_pyml.ml` and `venv_env_gym_pyml.ml` under `reinforcement-learning` use Python objects whose verification is not the scope of this paper.

– `reinforcement-learning/policy_gradient.ml` uses mutable lists which cannot be replaced with another datatype already supported in GraTen.

– `yolo/darknet.ml` and `translation/lang.ml` use hash tables which are not supported in GraTen yet.

– `translation/dataset.ml` and `translation/lang.ml` are irrelevant as tensor objects do not appear in them.

Evaluation We evaluated the best-effort inference of GraTen on the following three aspects.

First, we counted the assertions inserted into the original program when GraTen is used for the target program. Since the assertions indicate the program points that could fail at runtime, the user of GraTen would wish to pay attention to the location and the number of inserted assertions and try to decrease them.

Second, we counted the minimum number of type annotations required to type-check the program with minimum assertions inserted. This is for evaluating the realistic programmers' burden of trying to statically verify the program with type annotations. The annotations were added in such a way that the types of the functions do not lose the original generality. The type annotations are counted by the number of refinement types with non-`true` refinement predicates in them. For example, the following annotation counts as 3 because the refinement of the input tensor and the two output tensors are not `true`, but the refinement of the annotation of the second argument `bool` is `true`.

```
tensor([x]) -> ~is_training:bool -> tensor([x]) * tensor([x])
```

Third, we also measured the time taken by GraTen to analyze the unannotated and annotated programs. The experiments were conducted on a Linux machine with 12-core Intel i5-11400 (2.60GHz) and GraTen is implemented in Haskell with GHC version 9.0.2.

5.2 Experimental Results

Table 1 summarizes the experimental results. We analyze those results by the following three aspects: assertions, type annotations and analysis time.

Inserted Assertions Out of the 26 programs tested, 10 programs required no type annotations to type-check without assertions, and other 7 programs type-checked without assertions after adding appropriate type annotations. For the remaining 9 programs such as gan/began.ml and gan/gan_stability.ml, we could not eliminate all assertions, although some of them were removed after adding type annotations. The remaining assertions were due to the imprecise type signatures of some library functions. For instance, `Torch.Serialize.load` is a function that loads a tensor from a file and its type signature is defined as follows.

```
val load : ~filename:string -> tensor
```

The return type of `load` is simply defined as `tensor` since it is impossible to assume any properties about its shape. As a result, an assertion was inserted to check if the loaded tensor satisfies the requirement to run the program without uncaught errors. Even adding type annotations to the loaded tensor does not remove the assertion.

Some other functions are given imprecise types due to GraTen's immature support of polymorphic data types. For example, the type of `Tensor.stack` is defined as follows because GraTen does not effectively support non-integer lists yet. Refining the return types of such functions is left as future work.

```
val stack : ~dim:int -> list (tensor) -> tensor
```

Location under examples/	LOC	Unannotated		Annotated		
		time (s)	#assert	#annot	time (s)	#assert
char_rnn/char_rnn.ml	98	1.647	1	2	0.664	0
cifar/cifar_train.ml	72	0.311	0	-	-	-
cifar/densenet.ml	116	2.603	6	2	1.304	0
cifar/fast_resnet.ml	64	0.293	0	-	-	-
cifar/preact_resnet.ml	85	2.535	8	5	0.346	0
cifar/resnet.ml	78	2.597	8	4	0.396	0
gan/began.ml	220	1.581	1	-	-	-
gan/gan_stability.ml	224	4.441	40	2	1.410	2
gan/mnist_cgan.ml	117	0.498	1	-	-	-
gan/mnist_dcgan.ml	136	1.418	4	2	0.500	0
gan/mnist_gan.ml	83	0.308	0	-	-	-
gan/progressive_growing_gan.ml	118	0.734	0	-	-	-
gan/relativistic_dcgan.ml	171	0.659	1	-	-	-
jit/load_and_run.ml	16	0.214	1	-	-	-
min-gpt/mingpt.ml	207	3.036	8	6	2.686	0
mnist/conv.ml	53	0.250	0	-	-	-
mnist/linear.ml	50	0.235	0	-	-	-
mnist/nn.ml	39	0.210	0	-	-	-
pretrained/finetuning.ml	69	0.294	0	-	-	-
pretrained/predict.ml	68	0.303	2	-	-	-
reinforcement-learning/a2c.ml	105	0.418	0	-	-	-
reinforcement-learning/ppo.ml	129	0.438	0	-	-	-
reinforcement-learning/rollout.ml	91	0.734	9	5	0.425	1
translation/seq2seq.ml	258	3.800	11	34	1.023	3
vae/vae.ml	78	1.233	4	10	0.312	0
yolo/yolo.ml	144	1.027	4	1	0.985	3

Table 1. Results of running GraTen to the test cases. The second column is the size of the program after the modification. The third and fourth columns are the results for unannotated programs. The third column is the duration of the type-checking and the fourth column is the number of assertions inserted. From the fifth to the seventh columns are for the annotated programs. The fifth column is the number of annotations added to the program.

Patterns of Added Type Annotations As we added type annotations to the test cases, we observed that the program points that require type annotations have similarities. All of the type annotations fall into one of the following patterns.

(P1) Branches i.e., `if` expressions and `match` expressions with multiple branches (e.g., Figure 4 in Section 1).

(P2) Recursive functions. For example, `loop` in translation/seq2seq.ml is annotated as follows.

```
let rec loop
  :  ~state:tensor([1; enc.hidden_size])
  -> ~prevs:list ({ v:tensor | prod v.shape = 1 })
  -> ~max_length:int -> list ({ v:tensor | prod v.shape = 1 })
= fun ~state ~prevs ~max_length -> ...
```

(P3) Higher-order shape-polymorphic arguments. For example, `sample` in `char_rnn.ml` is annotated as follows.

```
let sample ~dataset ~lstm
  ~linear:(linear : x:{ v:tensor | last v.shape = hidden_size }
                 -> tensor(init x.shape @ [dataset.labels]))
  ~device = ...
```

(P4) Definition of record types. The current implementation of GraTen expects that the definition of record types describes the refinement types of each field.

(P5) Imprecise type signatures of primitive functions, or user-defined functions of dependent modules. For example, translation/seq2seq.ml has the following type annotation since the return type of `Tensor.stack` is only inferred to be `tensor` due to its imprecise type signature.

```
let enc_outputs : tensor([1; nth 1 v.shape; enc.hidden_size]) =
  Tensor.stack enc_outputs ~dim:1
```

The statically inferred type of `enc_outputs` here is `tensor([1; enc.hidden_size]) list`, so we would not need this type annotation if the type signature of `Tensor.stack` is appropriately defined. Since it is not possible to statically verify the correctness of these types of annotations, assertions would still be inserted after adding these annotations.

The first three patterns indicate that GraTen's current best-effort type inference does not effectively infer precise refinements for branches, recursive functions and higher-order shape-polymorphic arguments. The fourth pattern (P4) would be inevitable when using record types. It remains as future work to exempt users from having to add type annotations for (P5). With such improvements, we believe that it will become easier to find program points that require type annotations for better inference.

Number of Type Annotations There is no correlation between the number of assertions inserted into the unannotated program and the number of annotations needed to the program to minimize the number of assertions.

For example, adding two type annotations to `gan/gan_stability.ml` resulted in removing 38 assertions. This is because GraTen inferred an imprecise type for a helper function `resnet_block` without any type annotations, and it degraded the precision of the inference for the 24 callers of the function. Meanwhile, `translation/seq2seq.ml` required comparatively many type annotations as it has many definition of record types and several recursive functions with multiple inputs.

Analysis Time For all of the 11 annotated programs, GraTen's type checking for annotated programs was faster than the unannotated counterparts. This would be because having more static information made it easier for GraTen to infer more precise types and resolve more subsumption constraints easily.

5.3 Discussions

In this subsection, we discuss the strengths, weaknesses and our perspective on the future development of our system.

Performance of Best-Effort Inference As reported in the previous subsection, the best-effort inference of GraTen does not infer precise types for branches, recursions and higher-order shape-polymorphic arguments. While this may seem unsatisfying at a glance, the aim of this research is not to develop a perfect inference algorithm, but to propose a method that can work on unannotated programs and allows users to work interactively with the type checker to gradually add type annotations. With this respect, we believe that GraTen has achieved desirable results since it will be easy for the user to find out where to add type annotations. This is because (1) the inserted assertions can inform the user of the location of potential dynamic errors, and (2) all of the required type annotations would fall into one of the patterns listed in the previous section and thus should be predictable.

Lists of Tensors and Layers As of now, the refinement inference for lists in GraTen is limited to integer lists. Meanwhile, lists of tensors or lists of functions are commonly used in deep learning programs: `Tensor.cat` and `Tensor.stack` both take a list of tensors and return their concatenation, and `Layer.sequential` takes a list of layers (functions that take and return a tensor) and returns their composition.

A potential approach to support these library functions would be to add new refinement predicates for tensors lists or layer lists. For example, we can add a predicate $\texttt{composable}(x, S_1, S_2)$ which means that the composition of a list of layers x takes a tensor of shape S_1 and returns a tensor of shape S_2. The type of

`Layer.sequential` would be expressed with the shape polymorphic extension (see the full version [13]) as follows.

```
val sequential : forall S1 S2.
  { v:list(tensor -> tensor) | composable(x,S1,S2) }
                                -> tensor(S1) -> tensor(S2)
```

To practically infer `composable` predicate for layer lists, we would need to change the type-instantiated versions of list-manipulating functions as well. For instance, the type of the `cons` function for layers would need to be defined as follows.

```
val cons_layers
  :  forall S1 S2 S3. (tensor(S1) -> tensor(S2))
  -> { v:list(tensor -> tensor) | composable(v, S2, S3) }
  -> { v:list(tensor -> tensor) | composable(v, S1, S3) }
```

Reporting Incorrect Type Annotations Since our type system sees the standard refinement types as gradual, some users might find the behavior of GraTen unexpected in some cases. Consider the following function `f` which takes a matrix and returns a matrix obtained by transposing the input. Suppose that the programmer mistakenly annotated the return value of `f` to have the same shape as the input matrix.

```
let f x = (tr x : tensor(x.shape))
```

Although this type annotation does not hold in general, this program is not rejected by our type system because the annotation can hold if the input `x` is a square matrix. GraTen would output the following program with an assertion.

```
let f x = (fun y -> assert(y.shape = x.shape); y) (tr x)
```

To avoid such a situation, it would be possible to extend the type system with types with fully statically known refinements, and let the annotated types be interpreted as such.

6 Related Work

Tensor Shape Checking in Deep Learning Programs. The problem of tensor shape checking has been studied for decades by various contexts such as the numeric analysis [7,2] and the array-oriented languages with rank polymorphism [29,28,12]. Tensor shape checking for deep learning programs is still a new challenge because the shapes can be more complicated, and a variety of methods have been proposed both in academia and in industry.

Some tools statically check tensor shapes with advanced type systems. Hasktorch [3] is a Haskell binding of libtorch [20] which provides a mode that statically checks tensor shapes. Since they use the type-level programming feature of Haskell to implement the tensor shapes, tensor shapes are not first-class objects.

As a result, programs such as the one in Figure 1 cannot be expressed since it is impossible to define the function `f` whose type depends on the first-class object `s`. Relay [25,24] is an IR for deep learning compilers with a rich type system for tensor shape with type inference. Both Relay and Hasktorch support dynamic shape as a wild card in the static shape checking.

Apart from the type-based verification methods, some tensor shape error detection tools also take a static approach. Pythia [17,6] statically detects shape fault for TensorFlow [1] programs by keeping track of the tensor shapes throughout the program using value-flow analysis. The tracking of shape is in a best-effort manner, allowing the shape inference results to be "unknown" in some cases. The analysis crucially relies on the programming practice in TensorFlow to annotate tensor shapes as much as possible.

Other static checking tools took an approach that uses symbolic execution to collect constraints from the program and verifies it with a solver; Tensors Fitting Perfectly [21] and PyTea [15] are on this approach. Both methods remove loops from the program in an ad-hoc manner based on a reasonable assumption for the program.

Lastly, some took dynamic approaches to provide lightweight shape fault detection. ShapeFlow [31] is an abstract interpreter of TensorFlow programs; it shares the same APIs as TensorFlow but only calculates the shape of tensors. Users can run the analysis by replacing the import of TensorFlow with ShapeFlow in the target program, which executes more efficiently than the original TensorFlow program. Elichika [14] uses a similar method to ShapeFlow with a feature to display the interpreted shapes with a symbolic expression. These dynamic approaches enable quick analysis and require no type annotations, but provide no guarantee for untested inputs.

Static and Dynamic Checking for Refinement Types. Earlier work on dependent type system focused on decidable type checking and inference with restricted refinement logic [10,34,33,26]. Dynamic checking with contracts [19,9] offers expressive verification that cannot be covered with a static type system, but at a cost of runtime overhead. Naturally, the combination of static and dynamic checking has been actively explored by the successors of both parties.

Hybrid type checking [16], which our work is based on, extends the purely-dynamic method of using contracts by verifying specifications statically as much as possible. This method differs from ours in that it inserts a dynamic check only when the subtyping constraint is not proven to be valid or invalid. As a result, this method statically rejects the incorrectly annotated program that we discussed in Subsection 5.3, while our method accepts it with a dynamic check in the hope that a more precise type annotation will remove the need for a dynamic check. Our method can be understood as a variant of hybrid type checking with a focus on being gradual in adding type annotations.

The application of gradual typing to dependent type systems has also been studied [18,8]. Especially, gradual refinement types [18] is very similar to our type system in that it gradualizes only the predicate part of a refinement type system and the underlying simple type is static. One of the differences is that their system distinguishes statically-unknown refinement predicates with statically-known ones, while our system assumes that any refinement predicates can have a statically-unknown portion. For example, consider the following program:

$$\texttt{let}\ f\,x\,(y : \{\nu : \texttt{int} \mid \texttt{true}\}) = x/y$$

This program is rejected in their system because the type annotation of y indicates that the programmer is confident that y can be any integers including 0; otherwise, the type annotation should have been $\{\nu : \texttt{int} \mid \star\}$. Meanwhile, our system interprets the type annotation as not precise enough and accepts the program by inserting a dynamic check to y. Intuitively, $\{x : B \mid \varphi\}$ in our type system translates to $\{x : B \mid \varphi \wedge \star\}$ in gradual refinement types [18].

The type inference for gradual refinement types has been studied by Vazou et al. [30]. Their work restricts the refinement to liquid predicates [26] to maintain the decidability, while our work does not impose such a limitation.

7 Conclusion and Future Work

We presented an extension to the standard refinement type system which can be viewed as a gradual type system. The essence of this extension is the introduction of the consistent subtyping relation, which inserts to the source program assertions that checks statically-unverified properties at runtime. We also presented that the extended type system satisfies the refined criteria of gradual typing.

We then applied this type system for verifying tensor shapes with best-effort type inference. This application makes use of the property of the proposed type system that allows us to cover the limitation of the static best-effort analysis with dynamic checks. We also implemented a prototype type checker GraTen and applied it with some of the example programs publicly available in OCaml-Torch repository. We observed that, thanks to the best-effort type inference, users would not be required too many type annotations to statically type-check the whole program, and it would not be difficult to find where to add type annotations to improve the inference.

We conclude with some ideas for future work.

– Extension with type polymorphism. As we observed in the experiments, type polymorphic functions are frequently used in realistic programs. Extending our type system with ML-style type polymorphism would make the type checker more practical.

– Application for imperative languages with a dynamic type system, like Python. In this paper, we have chosen OCaml as the target of the prototype to ensure that the input program is statically-typed. Python would, however, be a more attractive target since it is widely used in the machine learning community.

Acknowledgments We would like to thank anonymous referees for useful comments. This work was supported by JSPS KAKENHI Grant Number JP20H05703.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)
2. Abe, A., Sumii, E.: A simple and practical linear algebra library interface with static size checking. arXiv preprint arXiv:1512.01898 (2015)
3. contributors, H.: Hasktorch. http://hasktorch.org/ (2020), [Online; accessed 15-July-2021]
4. contributors, O.T.: Ocaml-torch. https://github.com/LaurentMazare/ocaml-torch (2020), [Online; accessed 05-July-2021]
5. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
6. Dolby, J., Shinnar, A., Allain, A., Reinen, J.: Ariadne: analysis for machine learning programs. In: Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. pp. 1–10 (2018)
7. Eaton, F.: Statically typed linear algebra in haskell. In: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell. pp. 120–121 (2006)
8. Eremondi, J., Tanter, É., Garcia, R.: Approximate normalization for gradual dependent types. Proceedings of the ACM on Programming Languages **3**(ICFP), 1–30 (2019)
9. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming. pp. 48–59 (2002)
10. Freeman, T., Pfenning, F.: Refinement types for ml. In: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation. pp. 268–277 (1991)
11. Garcia, R., Clark, A.M., Tanter, É.: Abstracting gradual typing. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 429–442 (2016)
12. Gibbons, J.: Aplicative programming with naperian functors. In: Proceedings of the 1st International Workshop on Type-Driven Development. pp. 13–14 (2016)
13. Hattori, M., Kobayashi, N., Sato, R.: Gradual tensor shape checking. arXiv preprint arXiv:2203.08402 (2022)
14. Hattori, M., Sawada, S., Hamaji, S., Sakai, M., Shimizu, S.: Semi-static type, shape, and symbolic shape inference for dynamic computation graphs. In: Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. pp. 11–19 (2020)
15. Jhoo, H.Y., Kim, S., Song, W., Park, K., Lee, D., Yi, K.: A static analyzer for detecting tensor shape errors in deep neural network training code. arXiv preprint arXiv:2112.09037 (2021)
16. Knowles, K., Flanagan, C.: Hybrid type checking. ACM Trans. Program. Lang. Syst. **32**(2), 6:1–6:34 (2010). https://doi.org/10.1145/1667048.1667051, https://doi.org/10.1145/1667048.1667051
17. Lagouvardos, S., Dolby, J., Grech, N., Antoniadis, A., Smaragdakis, Y.: Static analysis of shape in tensorflow programs. In: 34th European Conference on Object-Oriented Programming (ECOOP 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2020)

18. Lehmann, N., Tanter, É.: Gradual refinement types. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 775–788 (2017)
19. Meyer, B.: Eiffel: the language. Prentice-Hall, Inc. (1992)
20. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in pytorch (2017)
21. Paszke, A., Saeta, B.: Tensors fitting perfectly. arXiv preprint arXiv:2102.13254 (2021)
22. Prabhu, S., Fedyukovich, G., Madhukar, K., D'Souza, D.: Specification synthesis with constrained horn clauses. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 1203–1217 (2021)
23. Rastogi, A., Swamy, N., Fournet, C., Bierman, G., Vekris, P.: Safe & efficient gradual typing for typescript. In: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 167–180 (2015)
24. Roesch, J., Lyubomirsky, S., Kirisame, M., Weber, L., Pollock, J., Vega, L., Jiang, Z., Chen, T., Moreau, T., Tatlock, Z.: Relay: A high-level compiler for deep learning. arXiv preprint arXiv:1904.08368 (2019)
25. Roesch, J., Lyubomirsky, S., Weber, L., Pollock, J., Kirisame, M., Chen, T., Tatlock, Z.: Relay: A new ir for machine learning frameworks. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. pp. 58–68 (2018)
26. Rondon, P.M., Kawaguci, M., Jhala, R.: Liquid types. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 159–169 (2008)
27. Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined criteria for gradual typing. In: 1st Summit on Advances in Programming Languages (SNAPL 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
28. Slepak, J., Manolios, P., Shivers, O.: Rank polymorphism viewed as a constraint problem. In: Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming. pp. 34–41 (2018)
29. Slepak, J., Shivers, O., Manolios, P.: An array-oriented language with static rank polymorphism. In: European Symposium on Programming Languages and Systems. pp. 27–46. Springer (2014)
30. Vazou, N., Tanter, É., Van Horn, D.: Gradual liquid type inference. Proceedings of the ACM on Programming Languages **2**(OOPSLA), 1–25 (2018)
31. Verma, S., Su, Z.: Shapeflow: Dynamic shape interpreter for tensorflow. arXiv preprint arXiv:2011.13452 (2020)
32. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: European Symposium on Programming. pp. 1–16. Springer (2009)
33. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation. pp. 249–257 (1998)
34. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 214–227 (1999)

A Type System for Effect Handlers and Dynamic Labels

Paulo Emílio de Vilhena(✉) and François Pottier

Inria, Paris, France
{paulo-emilio.de-vilhena,francois.pottier}@inria.fr

Abstract. We consider a simple yet expressive λ-calculus equipped with references, effect handlers, and dynamic allocation of effect labels, and whose operational semantics does not involve coercions or rely on type information. We equip this language with a type system that supports type and effect polymorphism, allows reordering row entries and extending a row with new entries, and supports (but is not restricted to) lexically scoped handlers. This requires addressing the issue of potential aliasing between effect names. Our original solution is to interpret a row not only as a permission to perform certain effects but also as a disjointness requirement bearing on effect names. The type system guarantees strong type soundness: a well-typed program cannot crash or perform an unhandled effect. We prove this fact by encoding the type system into a novel Separation Logic for effect handlers, which we build on top of Iris. Our results are formalized in Coq.

1 Introduction

Effect handlers [30,17] can be viewed as a generalization of exception handlers. Like raising an exception, *performing an effect* interrupts the normal flow of execution and transfers control to a handler. Unlike an exception handler, an effect handler gains access to a *delimited continuation*, which represents the fragment of the evaluation context comprised between the point where the effect was performed and the point where the effect handler was installed. Invoking this continuation resumes the computation whose execution was suspended by performing an effect.

To allow programmers to exploit several independent effects simultaneously, it is desirable for effects to have *names*. Each effect handler handles a specific name, or a specific set of names. When an effect is performed, the name of this effect determines which handler is selected. This idea immediately gives rise to several key questions about names. What are they: strings, variables, addresses? Where are they defined? What is their scope?

In the simplest approach [2,14,22], effect names are global. All possible names are predefined and are in scope everywhere. This approach is simple but unsatisfactory in terms of expressiveness and modularity: an accidental collision, where two unrelated pieces of code happen to use the same effect name, can have surprising unintended consequences. We illustrate this problem later on (§2).

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 225–252, 2023.
https://doi.org/10.1007/978-3-031-30044-8_9

To remedy this problem, several authors have proposed to change the nature of names. Their work falls broadly in two categories: the "lexical approach" and the "generative approach".

The "lexical approach" introduces local effect names with lexical scope. One can then think of an effect name essentially as a variable. *Tunneled exceptions* [42] and *lexically scoped handlers* [41,6,7,27] fall in this approach. In some of these proposals, the local effect name is never exposed to the user, but a "capability" to perform the effect is made available via a local variable. A potential pain point of this approach is that one must somehow ensure that a name or capability cannot escape its scope: this must be guaranteed by some combination of syntactic restrictions, runtime tests, and static typing rules.

The "generative approach" consists in allowing new effects to be generated afresh at runtime. This requires introducing a distinction between effect *labels*, which are allocated at runtime, and effect *names*, which are variables (with lexical scope) that the programmer uses to refer to effect labels. This is similar to the distinction between memory locations and variables that is traditionally used in the operational semantics of mutable references [29]. This approach has long been in use for exceptions in Standard ML [25] and OCaml [24], and is used also for effects in OCaml 5. It is powerful: in particular, it can simulate lexically scoped handlers.[1] However, it introduces several pitfalls of its own. First, it creates the possibility of nameless effects, that is, the possibility that there is no static effect name for a certain effect label. Second, it introduces the possibility of *aliasing* between effect names, that is, the possibility that two distinct effect names denote the same effect label. Aliasing creates a challenge for type system designers: if one cannot statically tell whether two effect names denote distinct labels, then it seems unclear how one can propose a sound and precise type discipline.

At least three ways of evading or addressing this challenge appear in the literature.

First, several mainstream languages adopt the generative approach but avoid the aliasing challenge by offering a weak type soundness guarantee: a well-typed program cannot crash, but can halt due to an unhandled exception or effect. This is the case in Standard ML, where exceptions are untracked, and in OCaml, where exceptions and effects are untracked. It is also the case in Eff [3].

Second, a number of authors evade or resolve the aliasing challenge by altering the syntax and the operational semantics of the language. Instead of letting the correspondence between an effect and a handler be determined purely by the notion of equality of effect labels or effect names, they introduce *coercions*

[1] This can be a source of confusion. A language that has "lexically scoped handlers" can, technically, be presented in either of these two styles. Biernacki et al. [6] present one semantics in each style, the "open semantics" and the "generative semantics", and prove an equivalence between them. Zhang and Myers [41] adopt what we believe is a combination of lexically scoped handlers and implicit arguments, which they refer to as "tunneling", in their surface language. This language is then translated down to a core language whose operational semantics is in the generative style.

that enable explicit disambiguation and collision avoidance. Examples include Koka [21] as well as several papers by Biernacki et al. [4,5].

Third, some authors evade the challenge by restricting the programming language in one or more ways, such as restricting attention to lexically scoped handlers [6,7] and forbidding first-class functions [7].

This sets the scene for this paper. We stick with the generative approach, which offers a simple and expressive semantics. We do not introduce coercions or otherwise alter the operational semantics. We do not restrict our attention to lexically scoped handlers. We address the aliasing challenge.

We propose TES, a type-and-effect system that statically rules out unhandled effects. As in most previous work, the potential effects of an expression are described by a *row*, a concept introduced to type-check records and variants [32,38] and later applied to the analysis of exceptions [28] and effects [14,22]. Type and effect polymorphism are supported. Furthermore, a simple and powerful subsumption relation allows reordering the entries in a row and extending a row with new entries, without any side conditions.

How is this possible? How is the aliasing challenge addressed? Our key idea is this: *whenever a question about aliasing arises, require absence of aliasing.* In other words, we interpret a row not just as a description of the names and types of the effects that may be performed, but also as *a requirement that these names be pairwise distinct.* For instance, if a typing judgment states that an expression e has effect $(s : \iota \Rightarrow \kappa) \cdot (s' : \iota' \Rightarrow \kappa')$, then this means not only that e may perform the effects s and s', but also that e *requires* the effect labels denoted by s and s' to be distinct. In the presence of effect polymorphism, if e has effect $(s : \iota \Rightarrow \kappa) \cdot \theta$, where θ is a row variable, then we take this to mean that e *requires* the effect label denoted by s to lie outside the set of effect labels denoted by θ. We adapt our typing and subtyping rules, where needed, so as to be sound with respect to this new interpretation of rows.

The reader may find our approach somewhat reminiscent of the manner in which the separating conjunction of Separation Logic [31] requires disjointness between the footprints of two formulae. Although this requirement may at first seem strong, experience has shown that Separation Logic is in fact concise and expressive. The examples that we present in Section 4.4 seem to suggest that our disjointness requirement is acceptable; we have not yet found examples where it is problematic. That said, we do not yet have practical experience with an implementation of this type system.

TES offers a strong type soundness guarantee: a well-typed program cannot crash and cannot halt due to an unhandled effect. To prove this fact, we follow a semantic approach that has become popular in the last few years [1,20,19]. We introduce TESLOGIC, a novel variant of Separation Logic, constructed on top of Iris [16], which allows reasoning about programs in the presence of effects and handlers, multi-shot continuations, and dynamic allocation of effect labels. We prove that this logic is sound, and we provide an interpretation of TES's typing rules in terms of TESLOGIC's reasoning rules. All of our results are formalized in Coq, and our Coq formalization is available [36].

In summary, the main contributions of this paper are the design of Tes, a type system for TesLang, a λ-calculus equipped with general references, effect handlers, and dynamic allocation of effect labels, and a proof of type soundness, which is carried out via a semantic interpretation into a new program logic, TesLogic.

In Section 2, we provide more background and examples about the semantics of effect handling: we discuss name collisions, effect coercions, lexically scoped handlers, and dynamic allocation of effect labels, and we justify why we wish to study a calculus where effect handling and dynamic allocation of effect labels are separate constructs. In Section 3, we present the syntax and operational semantics of TesLang. In Section 4, we introduce Tes and show a number of examples of constructions that Tes is able to type-check. In Section 5, we present a brief overview of the proof of type soundness. Finally, we discuss the related work and conclude.

2 A Panorama of Semantics for Effect Handlers

The various mechanisms that we have mentioned so far, namely lexically scoped handlers, dynamic allocation of effect labels, and effect coercions, aim to resolve the basic problem of accidental collisions between effect names. Let us illustrate this problem with an example.

Anticipating on Section 3, we use a λ-calculus equipped with constructs to perform and handle effects. The expression `perform` $s\ v$ *performs an effect* with effect name s and payload v. The expression `handle` e `with` s : $h \mid r$ installs an effect handler which monitors the execution of the subexpression e and which handles the effects that carry the name s.[2] If e returns a value v, then the *return branch* r is invoked and receives the value v as an argument. If e performs an effect with name s and with payload v, then the execution of e is suspended and control is transferred to the *effect branch* h, which receives the payload v and a continuation k representing the suspended computation.

Let us now introduce the function `bad_counter`. In a system of simple types, which does not keep track of effects, `bad_counter` expects a function *ff* of type $(\alpha \to \beta) \to \gamma$ and returns a function of type $(\alpha \to \beta) \to \gamma \times \texttt{int}$. The *intended* behavior of `bad_counter` *ff* is to produce a new function ff' such that ff' behaves like *ff* but at the same time counts how many times *ff* uses its argument. That is, for an arbitrary function f, the application $\mathit{ff}'\ f$ is expected to return a pair (v, n), where v is the result of the computation $\mathit{ff}\ f$ and n is the number of invocations of f that have taken place during this computation. The function `bad_counter` is defined as follows:

$$\texttt{bad\_counter}\ \mathit{ff} \;=\; \lambda f.\ \begin{pmatrix} \texttt{handle}\ \mathit{ff}\ (\lambda x.\ \texttt{perform}\ \mathit{tick}\ ();\ f\ x)\ \texttt{with} \\ \mathit{tick} : \lambda\_\ k.\ \lambda n.\ k\ ()\ (n+1) \mid \lambda y.\ \lambda n.\ (y, n) \end{pmatrix}\ 0$$

This code has a free effect name, *tick*. The function f is wrapped in a proxy which performs an effect named *tick*. This effect is handled by `bad_counter`; the

[2] For simplicity, this construct selects just one name, as opposed to a set of names.

handler implements a memory cell (in state-passing style) to count the number of ticks, that is, the number of calls made by *ff* to f.

Unfortunately, because this function uses a fixed effect name, *tick*, it can exhibit an unintended behavior, caused by an accidental collision of effect names. The following use of `bad_counter` exhibits this issue:

$$\texttt{bad\_counter}\ (\texttt{bad\_counter}\ (\lambda f.\, f\ ()))\ (\lambda\_.\, ())$$

Because the function $\lambda f.\, f\ ()$ calls its argument once, one might expect the above expression to return $(((), 1), 1)$. Its actual result, however, is $(((), 2), 0)$. In the interest of space, we omit an explanation of its operational behavior. The key reason why it behaves incorrectly is that *the two instances of* `bad_counter` *use the same effect name.* Each application of `bad_counter` installs a handler for the effect name *tick*. One handler is nested inside the other. As a result, the innermost handler intercepts two *tick* effects and the outermost handler never observes any effect, whereas what was naively intended was that each handler observes and handles one effect. As a result of the name collision, one of the effects is *accidentally handled* by the innermost handler.

To avoid or help avoid accidental collisions between names, the literature describes several mechanisms: (1) effect coercions, (2) lexically scoped handlers, which can be viewed as a restricted case of (3) dynamic allocation of effect labels. Let us now say a little more about these mechanisms.

Effect coercions. An effect coercion modifies the manner in which an effect is matched with one of the enclosing handlers. Perhaps the simplest example is that of the `lift` coercion [4,5], but there are other forms of coercions in the literature, such as `swap`. Normally, performing an effect named s transfers control to the innermost enclosing handler that selects the name s. However, in a language with effect coercions, if there is a `lift` coercion between the point where the effect is performed and the innermost enclosing handler, then this handler is *skipped* and control is transferred instead to the next enclosing handler for the name s.[3] Under such a semantics, a coercion can be employed to write a fixed version of `bad_counter`:

$$\begin{array}{l}\texttt{lift\_counter}\ \mathit{ff}\ = \\ \qquad \lambda f.\ \begin{pmatrix}\texttt{handle}\ \mathit{ff}\ (\lambda x.\, \texttt{perform}\ \mathit{tick}\ ();\, \texttt{lift}\ \mathit{tick}\ (f\ x))\ \texttt{with} \\ \mathit{tick} : \lambda\_\ k.\, \lambda n.\, k\ ()\ (n+1) \mid \lambda y.\, \lambda n.\, (y, n)\end{pmatrix}\ 0\end{array}$$

As desired, `lift_counter (lift_counter` $(\lambda f.\, f\ ()))\ (\lambda\_.\, ())$ returns the value $(((), 1), 1)$. One *tick* effect is intercepted by the innermost handler; the other effect is intercepted by the outermost handler thanks to the `lift` coercion. In Biernacki et al.'s λ^{HEL} [5], `lift_counter` is well-typed. The `lift` coercion is mandatory; without it, the code would be ill-typed.

3 A `lift` coercion behaves like an end-of-scope marker for the name s. This concept has been studied, independently of effects, by various authors [13,10].

Lexically scoped handlers and dynamic allocation of effect labels. Perhaps the most straightforward way to describe the operational behavior of lexically scoped handlers is by means of their encoding in terms of ordinary effect handlers and dynamic generation of effect labels. So, let us first extend our calculus with dynamic allocation of effect labels. We introduce the construct $\texttt{effect}\ s\ \texttt{in}\ e$, which binds the effect name s to a freshly generated *effect label*, then executes e. The effect name s is a local variable: its scope is the subexpression e. An effect label is a runtime entity; later in the paper, we let ℓ range over effect labels. In this setting, a "lexically scoped handler" is encoded (simulated) as follows:

$$\begin{aligned}&\texttt{lex-handle}\ e\ \texttt{with}\ h \mid r = \\ &\quad \texttt{effect}\ s\ \texttt{in}\ \texttt{handle}\ e\ (\lambda x.\ \texttt{perform}\ s\ x)\ \texttt{with}\ s : h \mid r\end{aligned} \tag{1}$$

This code first generates a fresh effect label, denoted by the name s. Then, it installs a handler for the name s. This handler monitors the execution of the expression e to the anonymous function $\lambda x.\ \texttt{perform}\ s\ x$, which can be viewed as a "capability" to perform the effect s.

A noteworthy aspect of the syntactic sugar $\texttt{lex-handle}\ e\ \texttt{with}\ h \mid r$ is that it does not explicitly involve any effect name. This construct is known as a "lexically scoped handler".

A lexically scoped handler can be used to write a fixed version of $\texttt{bad\_counter}$:

$$\texttt{counter}\ \mathit{ff} \ = \ \lambda f.\ \begin{pmatrix}\texttt{lex-handle}\ \lambda \mathit{tick}.\ \mathit{ff}\ (\lambda x.\ \mathit{tick}\ ();\ f\ x)\ \texttt{with} \\ \lambda\_\ k.\ \lambda n.\ k\ ()\ (n+1) \mid \lambda y.\ \lambda n.\ (y, n)\end{pmatrix}\ 0 \tag{2}$$

When $\texttt{lex-handle}$ is executed, a fresh effect label (which is never explicitly mentioned in this code) is generated. The variable *tick* stands for the "capability" to perform this fresh nameless effect. One can check that the expression $\texttt{counter}\ (\texttt{counter}\ (\lambda f.\ f\ ()))\ (\lambda\_.\ ())$ reduces to the value $(((), 1), 1)$, as desired, because the two instances of $\texttt{counter}$ generate two distinct dynamic labels and install one handler for each of these labels. Thus, no collision takes place.

Arguments in favor of dynamic allocation of effect labels. In summary, dynamic allocation of effect labels is a way of avoiding collisions between effect names. It can express lexically scoped handlers, but does not impose the use of lexically scoped handlers: it also allows working with global names when desired. Its dynamic semantics is simple. It is in use in several established programming languages, such as Standard ML and OCaml.

We believe that lexically scoped handlers are an elegant idiom, which is well suited to many but not all situations. So, we would not be satisfied with a restricted programming language where lexically scoped handlers are the sole form of effect handling. Indeed, lexically scoped handlers impose a somewhat unnatural "capability-passing" style, where the capability to perform an effect must be passed as an argument to a function (or captured in its closure). This style becomes especially cumbersome when multiple effects are involved. Implicit arguments can help, as suggested by Zhang and Myers [41] and by Odersky et al. [27]. However, elaboration of implicit arguments is usually a type-directed

$$\begin{array}{rl} n ::= & s \mid \ell \\ v ::= & () \mid \ell \mid \texttt{rec}\, f\, x.\, e \mid \S K \\ e ::= & v \mid x \mid e\; e \mid \texttt{ref}\; e \mid \texttt{!}\, e \mid e \coloneqq e \\ & \mid \texttt{effect}\; s\; \texttt{in}\; e \mid \texttt{perform}\; n\; e \mid \texttt{handle}\; e\; \texttt{with}\; n\; \texttt{:}\; v \mid v \mid \texttt{eff}\; \ell\; v\; K \\ K ::= & \bullet \mid e\; K \mid K\; v \mid \texttt{ref}\; K \mid \texttt{!}\, K \mid e \coloneqq K \mid K \coloneqq v \\ & \mid \texttt{perform}\; \ell\; K \mid \texttt{handle}\; K\; \texttt{with}\; \ell\; \texttt{:}\; v \mid v \end{array}$$

Fig. 1. Syntax of effect values, values, expressions, and evaluation contexts

$$\begin{array}{c} \texttt{effect}\; s\; \texttt{in}\; e \;/\; \sigma \rightarrow e[\ell/s] \;/\; \sigma[\ell \mapsto ()] \\ \texttt{perform}\; \ell\; v \;/\; \sigma \rightarrow \texttt{eff}\; \ell\; v\; \bullet \;/\; \sigma \\ \texttt{handle}\; v\; \texttt{with}\; \ell\; \texttt{:}\; h \mid r \;/\; \sigma \rightarrow r\; v \;/\; \sigma \\ \texttt{handle}\; (\texttt{eff}\; \ell\; v\; K)\; \texttt{with}\; \ell\; \texttt{:}\; h \mid r \;/\; \sigma \rightarrow h\; v\; \S(\texttt{handle}\; K\; \texttt{with}\; \ell\; \texttt{:}\; h \mid r) \;/\; \sigma \\ \S K\; v \;/\; \sigma \rightarrow K[v] \;/\; \sigma \\ \\ (\texttt{eff}\; \ell\; v_1\; K)\; v_2 \;/\; \sigma \rightarrow \texttt{eff}\; \ell\; v_1\; (K\; v_2) \;/\; \sigma \\ e_1\; (\texttt{eff}\; \ell\; v_2\; K) \;/\; \sigma \rightarrow \texttt{eff}\; \ell\; v_2\; (e_1\; K) \;/\; \sigma \\ \texttt{handle}\; (\texttt{eff}\; \ell\; v\; K)\; \texttt{with}\; \ell'\; \texttt{:}\; h \mid r \;/\; \sigma \rightarrow \texttt{eff}\; \ell\; v\; (\texttt{handle}\; K\; \texttt{with}\; \ell'\; \texttt{:}\; h \mid r) \;/\; \sigma \end{array}$$

Fig. 2. The head reduction relation (selected rules)

translation. If at all possible, we wish to preserve the "type erasure" property: that is, we prefer a language whose operational semantics is not influenced by type information, because such a semantics is easier to explain to an end user. Similarly, we wish to avoid effect coercions because we believe that they introduce unwarranted complexity, making the language and its dynamic semantics more difficult to explain to programmers.

3 Syntax and Semantics

We introduce TesLang, a calculus with mutable state, effect handlers, multiple named effects, dynamic allocation of effect labels, and multi-shot continuations. The operational semantics of this calculus allows a continuation to be invoked several times. With respect to this semantics, the type system presented in this paper (§4) is strongly sound: it rules out all runtime errors (§5). With respect to a dynamic semantics where invoking a continuation twice causes a runtime failure, such as the semantics of OCaml 5, our type system would be weakly sound, because it does not rule out this kind of runtime failure. Ensuring that every continuation is invoked at most once would require an affine type system and is beyond the scope of this paper. We note that an affine program logic, such as Hazel [35], can guarantee that no continuation is invoked twice, therefore can guarantee strong soundness even in the presence of one-shot continuations.

Our small-step operational semantics is very straightforward. It is equipped with dynamic allocation of effect labels and with a standard treatment of effects and effect handlers [2]. When an effect with label ℓ is performed, a dynamic lookup takes place: the nearest enclosing handler that is able to handle the label ℓ is selected. This is expressed, in small-step style, via several reduction rules. In contrast with some papers in the literature, where *coercions* influence the process of selecting a handler [21,4,5], here, this process is based purely on equality of effect labels.

3.1 Syntax

We let f and x range over an infinite set of variables. We let s range over an infinite set of variables, and we refer to these variables as *effect names*. These two namespaces are independent of one another: an effect name cannot be passed as a parameter to a function. We let ℓ range over an infinite set of addresses. These addresses model both *memory locations* and *effect labels*. Both kinds of entities are dynamically allocated, so, for simplicity, we use a single namespace of addresses and a single store. Whereas variables f, x and effect names s can appear in source programs, memory locations and effect labels ℓ exist only at runtime. The reduction rules of the small-step semantics cause them to appear.

The syntax of effect values, values, expressions, and evaluation contexts is shown in Figure 1.

An *effect value* n is either an effect name s or an effect label ℓ. This syntactic category is closed under substitutions of effect labels for effect names. It is used in the constructs $\texttt{perform}\, n\, e$ and $\texttt{handle}\, e\, \texttt{with}\, n : v \mid v$. A programmer always writes $\texttt{perform}\, s\, e$ and $\texttt{handle}\, e\, \texttt{with}\, s : v \mid v$, where s is an effect name, but the more general form is required in the operational semantics.

A *value* v is the unit value $()$, a memory location ℓ, a possibly recursive function $\texttt{rec}\, f\, x.\, e$, or a continuation $\S K$.

The syntax of *expressions* e includes values, variables, function application, operations for allocating, reading, and writing references, as well as constructs for allocating a fresh effect label, performing an effect, and handling an effect. Sequencing is encoded as function application: $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ is sugar for $(\lambda x.\, e_2)\ e_1$. The construct $\texttt{effect}\ s\ \texttt{in}\ e$ dynamically allocates a new effect label and binds the effect name s to this label in the expression e. The construct $\texttt{perform}\ s\ v$ performs an effect whose name is s and whose *payload* is the value v. The construct $\texttt{handle}\, e\, \texttt{with}\, s : h \mid r$ monitors the execution of the expression e. If an effect named s is performed, then the effect branch h takes control. If a value is returned, then the return branch r takes control. An effect that carries a name other than s is propagated up through this construct. Finally, the construct $\texttt{eff}\ \ell\ v\ K$, an *active effect*, does not appear in source program, but plays a role in the operational semantics, as we shall explain in the next subsection.

Our Coq formalization [36] covers a richer calculus, whose features include base types, pairs, sums, and lists.

The syntax of *evaluation contexts* K defines a right-to-left evaluation order. This choice is arbitrary: it is inspired by Iris's HeapLang language [33], but our results would hold also with left-to-right evaluation.

3.2 Semantics

The operational semantics of TesLang involves two relations, namely the *head reduction* relation $e \ / \ \sigma \rightarrow e' \ / \ \sigma'$ and the *reduction* relation $e \ / \ \sigma \longrightarrow e' \ / \ \sigma'$. They act on configurations, where a configuration $e \ / \ \sigma$ is a pair of an expression e and a *store* σ. The head reduction relation, a fragment of whose definition appears in Figure 2, is the most interesting relation. The reduction relation, whose definition is omitted, allows one step of head reduction to take place under an evaluation context.

A *store* is a finite map of addresses to values. We use addresses ℓ to denote both memory locations and effect labels. If ℓ denotes a memory location (that is, the address of a reference), then $\sigma(\ell)$ is the value stored at this address. If ℓ denotes an effect label, then the value $\sigma(\ell)$ is irrelevant: by convention, we use the unit value $()$.

The rules *not* shown in Figure 2, such as β_v-reduction and the rules for allocating, reading, and writing references, are standard.

The first rule in Figure 2 states that `effect` s `in` e allocates a fresh address ℓ, extends the store with a mapping of ℓ to the unit value, and substitutes the effect label ℓ for the effect name s in the expression e. (The rule has the side condition $\ell \notin \text{dom } \sigma$.) According to the second reduction rule, `perform` $\ell \ v$ reduces to an active effect `eff` $\ell \ v \bullet$. An active effect has the ability to capture the surrounding evaluation context, until it reaches a handler that is able to handle it. In this rule, it is initialized with an empty evaluation context $\bullet$. The last three rules in Figure 2 show how an active effect captures its evaluation context, one frame at a time. (The last rule has the side condition $\ell \neq \ell'$.) The third and fourth rules in Figure 2 show how the return branch or the effect branch of a `handle` construct are taken. In the latter rule, the handler h is applied to the payload value v and to a continuation, which reifies the captured evaluation context K. The continuation contains a copy of the effect handler: this is a *deep-handler semantics* [15]. The fifth reduction rule in Figure 2 describes the application of a continuation $\S K$ to a value v.

4 Type System

4.1 Syntax of types, rows, and signatures

We let α, β, and γ range over an infinite set of *type variables*. We let θ range over an infinite set of *row variables*. We distinguish three syntactic categories, namely *types*, *rows*, and *signatures* (Figure 3). The syntax of types is stable under substitutions of types τ for type variables α. The syntax of rows is stable under substitutions of rows ρ for row variables θ, for an ad hoc notion of substitution, which reduces row concatenation expressions "$\rho \cdot \rho'$" on the fly.[4]

[4] The distinction between rows and signatures enforces the view that a row ρ is a list where each component (known as a "signature") is either a signature for an effect name s or a row variable θ. Thus, we impose a simple form on rows. As an alternate

$$\begin{aligned}\tau,\ \kappa,\ \iota &::= \texttt{unit} \mid \bot \mid \top \mid \alpha \mid \tau\ \texttt{ref} \mid \tau \xrightarrow{\rho} \tau \mid \forall\alpha.\,\tau \mid \forall\theta.\,\tau\\ \rho &::= \langle\rangle \mid \sigma\cdot\rho\\ \sigma &::= (s : \tau \Rightarrow \tau) \mid \theta\end{aligned}$$

Fig. 3. Syntax of types, rows, and signatures

$$\textsc{Sub}\quad \frac{\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau \qquad \rho' \vdash_b \rho \leq_R \rho' \qquad \rho' \vdash \tau \leq_T \tau'}{\Xi \mid \Delta \mid \Gamma \vdash e : \rho' : \tau'}$$

$$\textsc{Var}\quad \frac{\Gamma(x) = \tau}{\Xi \mid \Delta \mid \Gamma \vdash x : \rho : \tau}$$

$$\textsc{RecFun}\quad \frac{\Xi \mid \Delta \mid \Gamma, f : \tau \xrightarrow{\rho} \kappa, x : \tau \vdash e : \rho : \kappa}{\Xi \mid \Delta \mid \Gamma \vdash \texttt{rec}\, f\, x.\, e : \langle\rangle : \tau \xrightarrow{\rho} \kappa}$$

$$\textsc{App}\quad \frac{\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau \xrightarrow{\rho} \kappa \qquad \Xi \mid \Delta \mid \Gamma \vdash e' : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash e\, e' : \rho : \kappa}$$

$$\textsc{TypeIntro}\quad \frac{\alpha \notin \Xi, \Gamma, \rho \qquad \Xi, \alpha \mid \Delta \mid \Gamma \vdash v : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash v : \rho : \forall\alpha.\,\tau}$$

$$\textsc{TypeElim}\quad \frac{\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \forall\alpha.\,\tau}{\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau[\tau'/\alpha]}$$

$$\textsc{RowIntro}\quad \frac{\theta \notin \Xi, \Gamma, \rho \qquad \Xi, \theta \mid \Delta \mid \Gamma \vdash v : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash v : \rho : \forall\theta.\,\tau}$$

$$\textsc{RowElim}\quad \frac{\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \forall\theta.\,\tau}{\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau[\rho'/\theta]}$$

$$\textsc{Effect}\quad \frac{s \notin \Gamma, \rho, \tau \qquad \Xi \mid \Delta, s \mid \Gamma \vdash e : (s : \texttt{abs}) \cdot \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash \texttt{effect}\ s\ \texttt{in}\ e : \rho : \tau}$$

$$\textsc{Perform}\quad \frac{s \in \Delta \qquad (s : \iota \Rightarrow \kappa) \in \rho \qquad \Xi \mid \Delta \mid \Gamma \vdash e : \rho : \iota}{\Xi \mid \Delta \mid \Gamma \vdash \texttt{perform}\ s\ e : \rho : \kappa}$$

$$\textsc{Handle}\quad \frac{\begin{array}{c} s \in \Delta \qquad \Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau \\ \rho = (s : \iota \Rightarrow \kappa) \cdot \rho_0 \qquad \rho' = (s : \iota' \Rightarrow \kappa') \cdot \rho_0 \\ \Xi \mid \Delta \mid \Gamma \vdash h : \rho' : \iota \to (\kappa \xrightarrow{\rho'} \tau') \xrightarrow{\rho'} \tau' \qquad \Xi \mid \Delta \mid \Gamma \vdash r : \rho' : \tau \xrightarrow{\rho'} \tau' \end{array}}{\Xi \mid \Delta \mid \Gamma \vdash \texttt{handle}\ e\ \texttt{with}\ s : h \mid r : \rho' : \tau'}$$

Fig. 4. The type system (selected rules)

Our types are standard: they include the unit type `unit`, the bottom and top types $\bot$ and $\top$, type variables α, reference types, effect-annotated arrow types, value-polymorphic types, and effect-polymorphic types. Effect-annotated arrow types and effect-polymorphic types are discussed below.

A row is a list of *signatures* σ. A signature, in turn, is either a *singleton signature* $s : \iota' \Rightarrow \kappa'$ or a *row variable* θ. A singleton signature $s : \iota' \Rightarrow \kappa'$ means that performing the effect s is permitted and is analogous to calling a function of argument type ι' and return type κ'. According to this reading, a singleton signature of the form $s : \bot \Rightarrow \top$ actually forbids the effect s, because a function whose argument type is $\bot$ can never be called. We write $s : \mathtt{abs}$ as a short-hand for this signature, and we refer to it as an *absence signature* for the effect s.

In addition to an argument type τ and a return type κ, an arrow type $\tau \xrightarrow{\rho} \kappa$ carries an "effect", that is, a row ρ. Intuitively, a value of type $\tau \xrightarrow{\rho} \kappa$ is a function, which, when applied to an argument of type τ, either returns a result of type κ or performs an effect that is permitted by the row ρ. On top of this standard reading of effect annotations, Tes introduces a novel aspect. The effect annotation ρ is interpreted not only as a set of permitted effects, but also as a precondition: we impose the semantic requirement that *a function of type $\tau \xrightarrow{\rho} \kappa$ can be invoked only if the multiset of effect labels denoted by the row ρ has no duplicate elements.* This is not a syntactic requirement, which would be either "true" or "false" and would be decided just by inspecting the syntax of the row ρ. Indeed, in general, a row contains occurrences of effect names s, which denote a-priori-unknown effect labels, and of row variables θ, which denote a-priori-unknown multisets of effect labels. What we wish to require is that, *at runtime, after effect names and row variables have been substituted away* by some substitution η, a function of type $\tau \xrightarrow{\rho} \kappa$ can be invoked only if no effect label appears twice in the closed row $\eta(\rho)$. Thus, the requirement that "ρ contains no duplicate labels" should be thought of as a *disjointness hypothesis* bearing on the row ρ. Such a hypothesis may or may not be satisfied, depending on how the effect names and row variables that occur in ρ are instantiated.

In Tes, disjointness hypotheses are sometimes explicit and most of the time implicit. In the subsumption judgments (Figure 5), a disjointness context D is explicit: it can be interpreted as a conjunction of disjointness hypotheses. In function types $\tau \xrightarrow{\rho} \kappa$ and in typing judgments $\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau$, an *implicit disjointness hypothesis* bearing on the row ρ is built in, so there is no need for an explicit disjointness context.

An effect-polymorphic type $\forall\theta.\ \tau$ involves a universal quantification over a row variable θ. For instance, the function `iter`, which iterates over a list, can be defined as follows:

$$\mathtt{iter} = \mathtt{rec}\ iter\ xs\ f.\ \mathtt{match}\ xs\ \mathtt{with}\ (\lambda x\ xs.\ f\ x\mathtt{;}\ iter\ xs\ f \mid \lambda\_.\ ()) \qquad (3)$$

path, one could use a single syntactic category $\rho ::= \langle\rangle \mid \rho \cdot \rho \mid (s : \tau \Rightarrow \tau) \mid \theta$, where a more general form of row concatenation is allowed. This would allow using a standard notion of substitution, and would lead to different statements for some of the row subsumption rules.

This function admits the following value- and effect-polymorphic type:

$$\texttt{iter} : \forall\alpha.\ \forall\theta.\ \alpha\ \texttt{list} \xrightarrow{} (\alpha \overset{\theta}{\rightarrow} \texttt{unit}) \overset{\theta}{\rightarrow} \texttt{unit}$$

This type states that the call `iter` xs f is safe, regardless of what the elements of the list xs might be, and regardless of what effects the user function f might perform. This type also guarantees that `iter` does not perform any effect of its own: instantiating θ with $\langle\rangle$ shows that this must be the case. Finally, one might think that this type guarantees that `iter` cannot intercept the effects performed by f. This may or may not be true, depending on which interpretation of effect-polymorphic types is chosen. A stronger interpretation can guarantee this property, but rules out certain useful programming language constructs, such as "dynamic-wind". Conversely, a weaker interpretation of effect-polymorphic types allows type-checking "dynamic-wind", but breaks this guarantee. At this time, the interpretation that we have verified in Coq is the weaker one (§5). We further discuss this point in Section 6.

4.2 The typing judgment

A typing judgment in TES takes the form $\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau$. It involves three environments: a *row- and type-variable context* Ξ, which binds row and type variables θ and α; an *effect-name context* Δ, which binds effect names s; and a *type environment* Γ, which maps variables x to types τ. This typing judgment states that the expression e has effect ρ and type τ. Like an arrow type, this judgment involves an implicit disjointness hypothesis bearing on the row ρ. That is, this judgment guarantees that it is safe to execute e *provided* the row variables and type variables in Ξ are instantiated in such a way that the multiset of effect labels denoted by ρ has no duplicate elements.

A selection of the typing rules appears in Figure 4. The typing rules for variables, functions, and applications are the same as in most type-and-effect systems. The typing rules for references are also standard, and are omitted. The rules TYPEINTRO, TYPEELIM, ROWINTRO, ROWELIM, which introduce and eliminate value- and effect-polymorphic types, are also standard. In the presence of mutable state, an unrestricted introduction rule for polymorphic types is unsound [34]. In this paper, we avoid this problem simply by building the value restriction [39,12] into TYPEINTRO and ROWINTRO. Our Coq formalization [36] proposes a more elaborate approach, where function types and typing judgments are annotated with *purity attributes*. This approach yields a slightly more expressive system, where, in particular, `perform` s x is considered a pure expression, therefore can receive a polymorphic type.

Rule EFFECT, read from bottom to top, changes the current effect from ρ to $(s : \texttt{abs}) \cdot \rho$. Intuitively, this means several things. First, while type-checking e, it is safe to *assume* that the effect label denoted by s is disjoint from the multiset of effect labels denoted by ρ. This assumption is implicitly expressed by the mere appearance of the row $(s : \texttt{abs}) \cdot \rho$ in the premise. This assumption is justified indeed, since the effect name s is bound to a *fresh* effect label when `effect` s `in` e

is executed. Second, because of the absence signature $s : \mathtt{abs}$, one must *check* that the expression e does not perform any effect with the name s. This seems a natural and unavoidable restriction: if such an effect was allowed, there would be no static effect name by which it can be described. Third, because of the side condition $s \notin \rho$, one must *check* that the row that appears in the premise contains *at most one singleton signature* for the effect name s. As a counter-example, if the expression e has effect $(s : \mathtt{abs}) \cdot (s : \mathtt{abs})$, then the typing rule Effect cannot be applied. The subsumption rule Sub cannot help, because the subsumption judgment $(s : \mathtt{abs}) \cdot (s : \mathtt{abs}) \leq (s : \mathtt{abs})$ does not hold. Thus, the rule Effect enforces a disjointness constraint.

Rule Perform states that, when one performs an effect whose signature is $s : \iota \Rightarrow \kappa$, one must pass a payload value of type ι, and, in return, one can expect a value of type κ. This supports the intuitive idea that performing an effect is analogous to calling an effect-free function of type $\iota \rightarrow \kappa$.

Rule Handle type-checks $\mathtt{handle}\ e\ \mathtt{with}\ s : h \mid r$, where the expression e is monitored by a handler for the effect s. This rule expresses the idea that this construct establishes a boundary between the inside, where effects named s may be performed in accord with the signature $s : \iota \Rightarrow \kappa$, and the outside, where effects named s may be performed in accord with a different signature $s : \iota' \Rightarrow \kappa'$. Because $s : \mathtt{abs}$ is sugar for $s : \bot \Rightarrow \top$, this rule also covers the common case where the effect s is absent on the outside. Both the effect branch h and the return branch r are part of the “outside world”, so their effects are described by the outside row ρ'. This remark explains all occurrences of ρ' in the last two premises, except the one in the type of the continuation. The continuation, which is the second parameter of the effect branch h, has type $\kappa \xrightarrow{\rho'} \tau'$. Because we have adopted a “deep-handler” semantics (§3), a copy of the handler is reinstalled inside the continuation. This explains why the effect ρ' and the result type τ' of the continuation are the same as those of the whole $\mathtt{handle}$ construct.

Rule Sub weakens a typing judgment by replacing an effect ρ and a type τ with a weaker effect ρ' and a weaker type τ'. This rule relies on several subsumption judgments, which we discuss next.

4.3 The subsumption judgments

The subsumption judgments on types, signatures, and rows appear in Figure 5. An original aspect is that these judgments depend on a *disjointness context* D, which appears on the left of the turnstile. A disjointness context is a (possibly empty, unordered) list of rows, and is interpreted as a conjunction of disjointness hypotheses: one hypothesis bears on each row. For instance, the disjointness context $(s_1 : \iota_1 \Rightarrow \kappa_1) \cdot (s_2 : \iota_2 \Rightarrow \kappa_2)$, $(s_3 : \iota_3 \Rightarrow \kappa_3) \cdot \theta$, which is a list of two rows, is equivalent to a conjunction of two disjointness hypotheses. The first hypothesis is equivalent to $s_1 \neq s_2$: it represents the assumption that the effect names s_1 and s_2 denote two distinct effect labels. The second hypothesis expresses the assumption that the effect label denoted by s_3 is not a member of the multiset of effect labels denoted by θ *and* that this multiset has no duplicate elements.

Type subsumption

TypeRefl
$$D \vdash \tau \leq_T \tau$$

Bot
$$D \vdash \bot \leq_T \tau$$

Top
$$D \vdash \tau \leq_T \top$$

TypeTrans
$$\frac{D \vdash \tau \leq_T \tau' \qquad D \vdash \tau' \leq_T \tau''}{D \vdash \tau \leq_T \tau''}$$

Arrow
$$\frac{D, \rho' \vdash \tau' \leq_T \tau \qquad D, \rho' \vdash_b \rho \leq_R \rho' \qquad D, \rho' \vdash \kappa \leq_T \kappa'}{D \vdash \tau \xrightarrow{\rho} \kappa \leq_T \tau' \xrightarrow{\rho'} \kappa'}$$

Signature subsumption

SigRefl
$$D \vdash \sigma \leq_S \sigma$$

SigCons
$$\frac{D \vdash \iota \leq_T \iota' \qquad D \vdash \kappa' \leq_T \kappa}{D \vdash (s : \iota \Rightarrow \kappa) \leq_S (s : \iota' \Rightarrow \kappa')}$$

Row subsumption

Empty
$$D \vdash_b \langle\rangle \leq_R \langle\rangle$$

Extend
$$D \vdash_b \rho \leq_R \sigma \cdot \rho$$

Swap
$$D \vdash_b \sigma \cdot \sigma' \cdot \rho \leq_R \sigma' \cdot \sigma \cdot \rho$$

RowCons
$$\frac{\begin{array}{c} D \vdash \sigma \leq_S \sigma' \\ D \vdash_{false} \rho \leq_R \rho' \end{array}}{D \vdash_b \sigma \cdot \rho \leq_R \sigma' \cdot \rho'}$$

Erase
$$\frac{D \Vdash s \# \rho}{D \vdash_{true} (s : \mathsf{abs}) \cdot \rho \leq_R \rho}$$

RowTrans
$$\frac{\begin{array}{c} D \vdash_b \rho \leq_R \rho' \\ D \vdash_b \rho' \leq_R \rho'' \end{array}}{D \vdash_b \rho \leq_R \rho''}$$

Effect/row disjointness

$$D \Vdash s \# \langle\rangle$$

$$\frac{D \Vdash s \# \sigma \qquad D \Vdash s \# \rho}{D \Vdash s \# (\sigma \cdot \rho)}$$

$$\frac{\rho \in D \qquad \{(s : \cdot \Rightarrow \cdot),\ (s' : \cdot \Rightarrow \cdot)\} \subseteq_m \rho}{D \Vdash s \# (s' : \iota' \Rightarrow \kappa')}$$

$$\frac{\rho \in D \qquad \{(s : \cdot \Rightarrow \cdot),\ \theta\} \subseteq_m \rho}{D \Vdash s \# \theta}$$

Fig. 5. The subsumption judgments

In the subsumption rules, the disjointness context is extended in the rule Arrow and exploited in the rule Erase. Elsewhere, it is just transported.

Subsumption on types. The subsumption judgment on types $D \vdash \tau \leq_T \tau'$ means that, under the hypothesis D, τ is a subtype of τ'. The rules in Figure 5 state that this relation is reflexive, transitive, and admits $\bot$ and $\top$ as bottom and top elements. On function types, as usual, subsumption is contravariant in the domain and covariant in the effect and in the codomain. One original aspect of Arrow is that this rule enriches the disjointness context: in the premises, the disjointness context changes from D to D, ρ'. The intuitive reason why this is sound is that if someone uses a function at type $\tau' \xrightarrow{\rho'} \kappa'$ then (at the point where the function is used) the disjointness hypothesis ρ' must be satisfied, because this hypothesis is part of our interpretation of function types. Thus, when proving that a function of type $\tau \xrightarrow{\rho} \kappa$ can be used as a function of type $\tau' \xrightarrow{\rho'} \kappa'$, it is safe to rely on the disjointness hypothesis ρ'.

Subsumption on signatures. The subsumption judgment on signatures takes the form $D \vdash \sigma \leq_S \sigma'$. Signature subsumption is reflexive and transitive. (Reflexivity is given by SigRefl; transitivity is derivable.) According to SigCons, unlike the standard function type constructor $\cdot \to \cdot$, the signature constructor $s : \cdot \Rightarrow \cdot$ is covariant in its domain and contravariant in its codomain. Indeed, when the signature $s : \iota \Rightarrow \kappa$ appears in the effect of an expression e, this means that e has *permission* to perform an effect named s at type $\iota \Rightarrow \kappa$. In other words, e can *assume* that performing an effect named s is analogous to calling a function of type $\iota \to \kappa$. This explains the reversed variance.

Subsumption on rows. The row subsumption judgment is $D \vdash_b \rho \leq_R \rho'$. The Boolean parameter b will be explained shortly. Row subsumption is reflexive and transitive. (Reflexivity is derivable; transitivity is given by RowTrans.) By combining Empty, Extend, RowCons, Swap, and RowTrans, one finds that if two rows, viewed as multisets of effect signatures, are related by multiset inclusion, then they are related by subsumption. Thus, subsumption allows permuting row entries in arbitrary ways and extending a row with new entries.

The last row subsumption rule, Erase, allows dropping an effect signature of the form $s : \texttt{abs}$. This rule may seem plausible because, both in the presence of the effect signature $s : \texttt{abs}$ and its absence, the effect s is forbidden. However, an unqualified axiom $\vdash (s : \texttt{abs}) \cdot \rho \leq_R \rho$ would be unsound. This is due to our interpretation of the row carried by a typing judgment (or by a function type) as a disjointness hypothesis. By changing a typing judgment that carries the row $(s : \texttt{abs}) \cdot \rho$ into one that carries the row ρ, *one removes the hypothesis* that the effect label denoted by s is not a member of the multiset of effect labels denoted by ρ. In order to safely remove a hypothesis, one must prove that it is satisfied. This explains why Erase must carry the premise $D \Vdash s \# \rho$, whose intuitive meaning is that "the hypotheses in D guarantee that the effect label denoted by s is not among the effect labels denoted by ρ".

The parameter b serves to forbid a use of Erase under RowCons. Erase requires this flag to be *true*, but RowCons sets it to *false* in its premise. Without this restriction, one could first combine Erase and DisjEmpty to prove $\vdash (s : \mathtt{abs}) \cdot \langle\rangle \leq_R \langle\rangle$, then use RowCons and induction to obtain $\vdash (s : \mathtt{abs}) \cdot \rho \leq_R \rho$ *without any side condition*, thus circumventing the side condition in Erase.

The four rules that define the effect/row disjointness judgment $D \Vdash s \# \rho$ are straightforward. The first two rules decompose the row ρ, which is a list of effect signatures σ. The last two rules look up the disjointness context D so as to find a disjointness hypothesis ρ that implies the goal. Whether ρ implies the goal is decided based on a simple syntactic criterion: the relation $\cdot \subseteq_m \cdot$ denotes multiset inclusion; the row on the right-hand side is viewed as a multiset of effect signatures.[5]

The desire to support Erase is the reason why the subsumption judgments carry a disjointness context. In a hypothetical simplified system where these judgments do not carry such a context, the premise of Erase would have to use an empty disjointness context *True*. This premise would become *True* $\Vdash s \# \rho$, which is false, so Erase would become inapplicable. Yet Erase is desirable, because it is useful in practice. We use it to type-check our encoding of a lexically scoped handler: this is illustrated in Section 4.4.

Why is $\vdash (s : \mathtt{abs}) \cdot \rho \leq_R \rho$ *unsound?* In the presence of this axiom, the judgment $\vdash (s : \mathtt{abs}) \cdot (s : \mathtt{abs}) \leq_R (s : \mathtt{abs})$ would be derivable. This judgment can be exploited to type-check the following unsafe program:

```
1 effect s in
2 handle
3   handle (perform s ()) with s : λx _. not x | λ_. true
4 with s : λ_ _. () | λ_. ()
```

This program is unsafe because the effect s is performed with a payload of type `unit`, namely the unit value () on line 3, and this effect is handled by the innermost handler, also on line 3, which expects the payload x to be a Boolean value. When this program is executed, it becomes stuck by attempting to execute the function application `not` ().

Yet, under the assumption $\vdash (s : \mathtt{abs}) \cdot (s : \mathtt{abs}) \leq_R (s : \mathtt{abs})$, this program is well-typed, with an empty row and with the type `unit`. Beginning at the root and working towards the leaves, the type derivation begins with an application of Effect, which changes the empty row into the row $(s : \mathtt{abs})$. Then, by using Sub and by exploiting the above assumption, the row $(s : \mathtt{abs})$ can be changed to $(s : \mathtt{abs}) \cdot (s : \mathtt{abs})$. At this point, the harm is done. Indeed, under the row $(s : \mathtt{abs}) \cdot (s : \mathtt{abs})$, the subprogram at lines 2–4 is well-typed. The fact that this row includes two signatures for the effect name s allows us to install two handlers for this name. The handler on line 2 allows its handlee—the expression

[5] Our Coq code [36] presently employs a different representation of disjointness contexts and a different definition of the effect/row disjointness judgment. We believe, but have not yet checked, that the Coq and paper formulations are equivalent.

on line 3—to perform effects according to the signature $s : \mathtt{unit} \Rightarrow \mathtt{unit}$. The handler on line 3 allows its handlee to perform effects as per $s : \mathtt{bool} \Rightarrow \mathtt{unit}$. The expression $\mathtt{perform}\ s\ ()$ is type-checked with respect to the composite row $(s : \mathtt{unit} \Rightarrow \mathtt{unit}) \cdot (s : \mathtt{bool} \Rightarrow \mathtt{unit})$, which means that this expression must respect *either* of these two signatures. It does indeed respect the first one, so it is well-typed.

4.4 Examples

Filter Recall the higher-order iteration function `iter` (Eq. 3), whose type is

$$\mathtt{iter} : \forall \alpha.\ \forall \theta.\ \alpha\ \mathtt{list} \to (\alpha \xrightarrow{\theta} \mathtt{unit}) \xrightarrow{\theta} \mathtt{unit}.$$

Let us use `iter` in the definition of `filter`:

$$\mathtt{filter}\ xs\ f = \mathtt{let}\ g = (\lambda x.\ \mathtt{if}\ f\ x\ \mathtt{then}\ \mathtt{perform}\ \mathit{yield}\ x)\ \mathtt{in}\ \mathtt{iter}\ xs\ g$$

The expression $\mathtt{filter}\ xs\ f$ "yields" each element x of the list xs in turn, by performing a *yield* effect if $f\ x$ returns `true`. In TES, `filter` is well-typed, and its type is:

$$\mathtt{filter} : \forall \alpha.\ \forall \theta.\ \alpha\ \mathtt{list} \to (\alpha \xrightarrow{\theta} \mathtt{bool}) \xrightarrow{(\mathit{yield} : \alpha \Rightarrow \mathtt{unit}) \cdot \theta} \mathtt{unit}$$

Checking that `filter` is well-typed is not difficult. Under the assumption that f has type $\alpha \xrightarrow{\theta} \mathtt{bool}$, the subexpression $f\ x$ has effect θ. Under the assumption that x has type α, the subexpression $\mathtt{perform}\ \mathit{yield}\ x$ has effect $(\mathit{yield} : \alpha \Rightarrow \mathtt{unit})$. Because our subsumption rules allow extending a row with a new entry and exchanging row entries, the composite subexpression $\mathtt{if}\ f\ x\ \mathtt{then}\ \mathtt{perform}\ \mathit{yield}\ x$ admits the composite effect $(\mathit{yield} : \alpha \Rightarrow \mathtt{unit}) \cdot \theta$.

What does `filter`'s type mean? Ostensibly, the row $(\mathit{yield} : \alpha \Rightarrow \mathtt{unit}) \cdot \theta$ tells us that every effect performed by $\mathtt{filter}\ xs\ f$ must be either a *yield* effect or an effect caused by f. Less obviously, these alternatives must be mutually exclusive: indeed, the row $(\mathit{yield} : \alpha \Rightarrow \mathtt{unit}) \cdot \theta$ carries the implicit requirement that the effect label denoted by *yield* is not among the effect labels denoted by θ. In other words, `filter`*'s type forbids f from performing yield effects.*

The reader may wonder what prevents us from instantiating θ with a row that includes the effect name *yield*, such as $(\mathit{yield} : \alpha \Rightarrow \mathtt{unit})$. The answer is, nothing prevents such an instantiation. The result, however, would be a view of `filter` as a function whose effect is $(\mathit{yield} : \alpha \Rightarrow \mathtt{unit}) \cdot (\mathit{yield} : \alpha \Rightarrow \mathtt{unit})$. Such an effect carries an unsatisfiable disjointness hypothesis, namely $\mathit{yield} \neq \mathit{yield}$. As a result, once the type of `filter` has been instantiated in this way, `filter` cannot be called anymore.[6]

[6] Technically, an application of this instantiated `filter` function can still be well-typed, but only if it appears in the body of a function which itself carries an unsatisfiable disjointness hypothesis and therefore can never be called.

Lexically scoped handlers We now derive a typing rule for lexically scoped handlers. Recall the encoding of a lexically scoped handler (Eq. 1):[7]

$$\begin{aligned}&\texttt{lex-handle}_s\ e\ \texttt{with}\ h \mid r = \\ &\qquad \texttt{effect}\ s\ \texttt{in handle}\ e\ (\lambda x.\, \texttt{perform}\ s\ x)\ \texttt{with}\ s : h \mid r\end{aligned}$$

For this construct, TES admits the following derived typing rule:

$$\textsc{LexHandle}\quad \frac{\begin{array}{c}\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \forall\theta.\,(\iota \xrightarrow{\theta} \kappa) \xrightarrow{\theta\cdot\rho} \tau \qquad s \notin \Gamma, \rho, \iota, \kappa, \tau, \tau' \\ \Xi \mid \Delta \mid \Gamma \vdash h : \rho : \iota \to (\kappa \xrightarrow{\rho} \tau') \xrightarrow{\rho} \tau' \qquad \Xi \mid \Delta \mid \Gamma \vdash r : \rho : \tau \xrightarrow{\rho} \tau'\end{array}}{\Xi \mid \Delta \mid \Gamma \vdash \texttt{lex-handle}_s\ e\ \texttt{with}\ h \mid r : \rho : \tau'}$$

This rule is similar to the typing rule for lexically scoped handlers that appears in Figure 3 of Biernacki et al.'s paper [6]. What is new and noteworthy is that we obtain this rule as a special case of a more permissive type discipline, TES, which supports general effect handlers, as opposed to just lexically scoped handlers.

In LEXHANDLE, whereas the effect on the outside is ρ, the effect on the inside is $\theta \cdot \rho$. That is, inside the handlee, one more effect is permitted. The handlee (the expression e) must be polymorphic in the row variable θ: that is, it must treat this extra effect as an abstract effect.

The derivation of LEXHANDLE involves an application of EFFECT and an application of HANDLE. While proving that the premises of HANDLE hold, a key step is to prove that the type of the effect branch h can be weakened as follows, where ρ' is a shorthand for $(s : \texttt{abs}) \cdot \rho$:

$$\frac{\Xi \mid \Delta \mid \Gamma \vdash h : \rho : \iota \to (\kappa \xrightarrow{\rho} \tau') \xrightarrow{\rho} \tau'}{\Xi \mid \Delta \mid \Gamma \vdash h : \rho : \iota \to (\kappa \xrightarrow{\rho'} \tau') \xrightarrow{\rho'} \tau'} \qquad \rho' = (s : \texttt{abs}) \cdot \rho$$

It is not at all obvious that this is possible! Two occurrences of ρ must be changed into ρ'. One occurrence is positive and one is negative, and the rows ρ and ρ' are not equal. Still, this implication can be established, via rule SUB. One must check the following chain of subsumption relations:

$$\iota \to (\kappa \xrightarrow{\rho} \tau') \xrightarrow{\rho} \tau' \leq_T \iota \to (\kappa \xrightarrow{\rho} \tau') \xrightarrow{\rho'} \tau' \leq_T \iota \to (\kappa \xrightarrow{\rho'} \tau') \xrightarrow{\rho'} \tau'$$

The first step requires $\vdash_b \rho \leq_R \rho'$, which, by EXTEND, is true. The second step requires $\rho' \vdash_{true} \rho' \leq_R \rho$, which, by ERASE, is true as well. The disjointness hypothesis ρ' plays a key role: indeed, $\mathit{True} \vdash_{true} \rho' \leq_R \rho$ is false. In other words, ERASE is applicable because the disjointness hypothesis ρ' is available, and this hypothesis exists because ARROW causes it to appear as it descends into the domains of two function types that are annotated with ρ'.

[7] This encoding requires choosing an arbitrary name s that does not occur in e, h or r. Furthermore, in the derivation of the typing rule LEXHANDLE, s may need to be renamed. On paper, we would normally not mention these details. However, because our Coq code does not currently allow α-conversion of effect names, we make s a parameter of the macro `lex-handle` and we include a freshness hypothesis bearing on s in LEXHANDLE.

Counter Using the type rule LexHandle, it is straightforward to check that `counter` (§2, Eq. 2) can be assigned the following type:

$$\texttt{counter} : \forall \alpha\, \beta\, \gamma.\quad (\forall \theta.\ (\alpha \xrightarrow{\theta} \beta) \xrightarrow{\theta} \gamma) \quad\rightarrow\quad \forall \theta.\ (\alpha \xrightarrow{\theta} \beta) \xrightarrow{\theta} (\gamma * \texttt{int})$$

This means that `counter` accepts an arbitrary effect-polymorphic second-order function ff and produces a function ff' whose type is similar to ff's type. The only difference between the types of ff and ff' is in their result types, to wit, γ versus $\gamma * \texttt{int}$.

It is not hard to see that the expression $\texttt{counter}\ (\texttt{counter}\ (\lambda f.\ f\ ()))\ (\lambda\_.\ ())$, where two instances of `counter` are nested, is also well-typed, and that its type is $(\texttt{unit} * \texttt{int}) * \texttt{int}$.

Mix The following second-order function, `mix`, involves a potentially challenging mixture of features:

$$\begin{array}{l} \texttt{mix}\ f\ \ = \\ \quad \texttt{handle}\ (\texttt{perform}\ s\ ();\ f\ ()) \\ \quad \texttt{with}\ s : \lambda\_\ k.\ k\ ()\ |\ \lambda\_.\ () \end{array}$$

The effect name s occurs free in this code, so this is *not* an instance of a lexically scoped handler. (We assume that the name s is introduced by the surrounding context.) The subexpression $\texttt{perform}\ s\ ();\ f\ ()$ visibly performs the effect s and calls the unknown function f, which itself may perform various effects, perhaps including the effect s. This subexpression is monitored by a handler for the effect s at type $\texttt{unit} \Rightarrow \texttt{unit}$.

In Tes, `mix` is well-typed. In fact, it admits several types. We show three: the first two are equivalent, and the last one subsumes the first two.

The first idea that comes to mind may be: "since f has an unknown effect, let's represent this effect with a row variable θ". Thus, one introduces a row variable θ, and one assumes that f has type $\texttt{unit} \xrightarrow{\theta} \texttt{unit}$. Under this assumption, one finds that $\texttt{perform}\ s\ ();\ f\ ()$ has effect $(s : \texttt{unit} \Rightarrow \texttt{unit}) \cdot \theta$. (The subsumption rule Extend is used, twice, to merge the effect of $\texttt{perform}\ s\ ()$ and the effect of $f\ ()$.) Finally, using Handle, one finds that the body of the function `mix` has effect $(s : \texttt{abs}) \cdot \theta$. In summary, `mix` admits the following type:

$$\texttt{mix} : \forall \theta.\ (\texttt{unit} \xrightarrow{\theta} \texttt{unit}) \xrightarrow{(s\,:\,\texttt{abs})\cdot\theta} \texttt{unit} \tag{4}$$

The effect $(s : \texttt{abs}) \cdot \theta$ carried by the second arrow means that `mix` never throws the effect s and transmits whatever effects f may throw, *provided these effects do not include* s. Indeed, the row $(s : \texttt{abs}) \cdot \theta$ is interpreted not only as a description of `mix`'s potential effects, but also as a disjointness constraint. Thus, the row $(s : \texttt{abs}) \cdot \theta$ in this type (4) cannot be replaced with just θ. Such a replacement would amount to discarding the disjointness constraint, which would be unsound.

The reader may wonder what happens if θ is instantiated, in the above type, with a row that mentions s, such as $s : \texttt{int} \Rightarrow \texttt{int}$. Technically, this is permitted, but yields a version of `mix` whose effect is $(s : \texttt{abs}) \cdot (s : \texttt{int} \Rightarrow \texttt{int})$. Such a function can never be called.

Thus, this type (4) effectively forbids f from performing effect s. One may wonder whether this fact can be made explicitly visible in the type of `mix`. In fact, it can. By the subsumption rules Arrow, Extend, and Erase, the type (4) is equivalent to the following type:

$$\texttt{mix} : \forall\theta.\ (\texttt{unit} \xrightarrow{(s\,:\,\texttt{abs})\cdot\theta} \texttt{unit}) \xrightarrow{(s\,:\,\texttt{abs})\cdot\theta} \texttt{unit} \tag{5}$$

Indeed, under the disjointness constraint carried by the outer arrow, the rows θ and $(s : \texttt{abs}) \cdot \theta$ are equivalent.

It is worth noting that this type allows the function f to use the effect s internally, if desired, and at an arbitrary type, provided this effect is handled internally by f and does not escape.

Finally, one may wonder whether it is necessary to forbid f from visibly performing effect s. In fact, it is not: one can allow f to perform this effect and let it escape, provided it is performed at type $\texttt{unit} \Rightarrow \texttt{unit}$, which is the type expected by the handler inside `mix`. It is not difficult to check that `mix` admits the following type:

$$\texttt{mix} : \forall\theta.\ (\texttt{unit} \xrightarrow{(s\,:\,\texttt{unit}\Rightarrow\texttt{unit})\cdot\theta} \texttt{unit}) \xrightarrow{(s\,:\,\texttt{abs})\cdot\theta} \texttt{unit} \tag{6}$$

This type (6) is in fact more general than (that is, a subtype of) the previous type (5). This follows directly from the fact that $s : \texttt{abs}$ is a short-hand for $s : \bot \Rightarrow \top$ and from the subsumption rules SigCons, RowCons, and Arrow.

5 Metatheory

In this section, we present the general architecture of the proof of our type soundness statement (Theorem 3), which states that, if a closed program e is well-typed, then e is *safe*: that is, e may diverge or terminate with a value, but cannot perform an unhandled effect. Full details are found in our Coq code [36].

Our first step is to interpret our typing judgments as *semantic typing judgments*. A semantic typing judgment $\Xi \mid \Delta \mid \Gamma \vDash e : \rho : \tau$ is a logical assertion stating that substituting *certain values* for the free variables of e yields a closed program that meets a *certain specification*. To fill in the details, one must define precisely which values may be substituted and what specification is met.

To do so, we introduce TesLogic, an extension of Iris [16], an expressive Separation Logic. Iris's base logic has no built-in support for effects and handlers, but allows constructing a program logic with such support. de Vilhena and Pottier define such a logic, Hazel [35]. Because Hazel is tailored for unnamed effects and one-shot continuations, we cannot re-use it. Nevertheless, in the design of TesLogic, we do rely on one of Hazel's key features, *protocols*.

A protocol Ψ describes a service on which the handlee can rely and which the handler must implement. Mathematically, it is a binary relation between a value v, the payload of the effect, and a predicate Φ, the precondition of the continuation for this effect. A typical example of a protocol is the *pre/post protocol*

Weakest precondition

$$\mathit{wp}\ e\ \langle E\rangle\{\Phi\} \triangleq \mathit{ValidDistinct}\ E.1 \mathrel{-\!\!*} \mathit{ewp}\ e\ \langle E\rangle\{\Phi\}$$

Basic weakest precondition

$$\begin{aligned}
\mathit{ewp}\ v\ \langle E\rangle\{\Phi\} &\triangleq \Phi(v)\\
\mathit{ewp}\ (\texttt{eff}\ \ell\ v\ K)\ \langle E\rangle\{\Phi\} &\triangleq \exists\Psi.(\ell,\Psi)\in E \ast (\uparrow_{\square}\Psi)\ v\ (\lambda w.\ \triangleright\,\mathit{ewp}\ K[w]\ \langle E\rangle\{\Phi\})\\
\mathit{ewp}\ e\ \langle E\rangle\{\Phi\} &\triangleq \forall\sigma.\ S(\sigma)\ {}^{\top}\!\!\Rrightarrow^{\emptyset}\\
&\quad\begin{cases}\exists e',\sigma'.\ e\,/\,\sigma \longrightarrow e'\,/\,\sigma' \ast\\ \forall e',\sigma'.\ e\,/\,\sigma \longrightarrow e'\,/\,\sigma'\ {}^{\emptyset}\!\!\Rrightarrow^{\emptyset}\triangleright\,{}^{\emptyset}\!\!\Rrightarrow^{\top}\\ \qquad S(\sigma') \ast \mathit{ewp}\ e'\ \langle E\rangle\{\Phi\}\end{cases}
\end{aligned}$$

Persistent upward closure

$$(\uparrow_{\square}\Psi)\ v\ \Phi \triangleq \exists\Phi'.\ \Psi\ v\ \Phi' \ast \square\forall w.\ \Phi'(w) \mathrel{-\!\!*} \Phi(w)$$

Validity-and-distinctness property

$$\mathit{ValidDistinct}\ L \triangleq \mathit{NoDup}\ L \wedge \bigwedge_{\ell\in L} \ell \mapsto_{\square} ()$$

Fig. 6. Definition of the weakest precondition

$\{\Phi_1\}.\{\Phi_2\}$, defined as $\lambda\,v\,\Phi.\ \Phi_1(v) \ast \square\forall\,w.\ \Phi_2(w) \mathrel{-\!\!*} \Phi(w)$. We use this protocol (in the interpretation of signatures, Figure 7) to attach a precondition Φ_1 and a postcondition Φ_2 to an effect: performing an effect with payload v is permitted if $\Phi_1(v)$ holds, and one can assume that it returns a value w such that $\Phi_2(w)$ holds. The symbol $\square$ is Iris's *persistence modality*. Here, it reflects the fact that continuations are multi-shot: a single `perform` expression can "return" several times with several different values of w, so we must be prepared to exploit Φ_2 several times.

To reason about labeled effects, we introduce the notion of a *protocol list* E, a list of pairs of a label and a protocol. Therefore, whereas Hazel's weakest precondition modality is parameterized with a single protocol, ours is parameterized with a protocol list. In our setting, the assertion $\mathit{wp}\ e\ \langle E\rangle\{\Phi\}$ means that (1) it is safe to execute e; (2) if e produces a value v then $\Phi(v)$ holds; and (3) if e performs an effect labeled ℓ then it does so according to a protocol Ψ such that $(\ell,\Psi)\in E$ holds. Its definition appears in Figure 6. It is broadly similar to Hazel's *wp* modality, save for three aspects: the use of a protocol list E; the use of a *persistent upward closure*; and the appearance of a *validity-and-distinctness property* as an assumption of the weakest precondition assertion. The persistent upward closure again has to do with the fact that continuations are multi-shot. The validity-and-distinctness property expresses two properties of the labels in the list E; first, these labels are pairwise distinct; second, these labels have been allocated. The latter fact is expressed by a *persistent points-to assertion* [37].

Interpretation of types (selected cases)

$$\mathcal{V}[\![\tau \xrightarrow{\rho} \kappa]\!]^{\delta}_{\eta}(v) \triangleq \square\,\forall\, w.\ \mathcal{V}[\![\tau]\!]^{\delta}_{\eta}(w) -\!\!* \ wp\ (v\ w)\ \langle \mathcal{R}[\![\rho]\!]^{\delta}_{\eta} \rangle \{\mathcal{V}[\![\kappa]\!]^{\delta}_{\eta}\}$$
$$\mathcal{V}[\![\forall \theta.\ \tau]\!]^{\delta}_{\eta}(v) \triangleq \forall\, E.\ \mathcal{V}[\![\tau]\!]^{\delta}_{\eta,\theta\mapsto E}(v)$$

Interpretation of rows and signatures

$$\mathcal{R}[\![\rho]\!]^{\delta}_{\eta} \triangleq \bigcup_{\sigma\in\rho} \mathcal{S}[\![\sigma]\!]^{\delta}_{\eta} \qquad \mathcal{S}[\![(s : \iota \Rightarrow \kappa)]\!]^{\delta}_{\eta} \triangleq (\delta(s),\ \{\mathcal{V}[\![\iota]\!]^{\delta}_{\eta}\}.\{\mathcal{V}[\![\kappa]\!]^{\delta}_{\eta}\})$$
$$\mathcal{S}[\![\theta]\!]^{\delta}_{\eta} \triangleq \eta(\theta)$$

Interpretation of typing judgments

$$\varXi \mid \varDelta \mid \varGamma \vDash e\ :\ \rho\ :\ \tau \triangleq \forall\, \eta,\ \delta,\ vs.\ \mathcal{G}[\![\varGamma]\!]^{\delta}_{\eta}(vs) -\!\!*\ wp\ (e[vs][\delta])\ \langle \mathcal{R}[\![\rho]\!]^{\delta}_{\eta} \rangle \{\mathcal{V}[\![\tau]\!]^{\delta}_{\eta}\}$$
$$\mathcal{G}[\![\varGamma]\!]^{\delta}_{\eta}(vs) \triangleq \forall\, \{x \mapsto \tau\} \subseteq \varGamma.\ \mathcal{V}[\![\tau]\!]^{\delta}_{\eta}(vs(x))$$

Fig. 7. Interpretation of types, rows, signatures, and typing judgments

This notion of *wp* enjoys a set of reasoning rules that we omit. The following theorem states that it is sound to reason about programs by means of these rules:

Theorem 1 (Soundness of TesLogic). *If $wp\ e\ \langle [\,] \rangle \{\varPhi\}$ holds, then e is safe.*

With TesLogic at hand, let us come back to the definition of the semantic judgment $\varXi \mid \varDelta \mid \varGamma \vDash e\ :\ \rho\ :\ \tau$.

As usual, a type τ is interpreted as a semantic type, that is, a persistent predicate $\mathcal{V}[\![\tau]\!]^{\delta}_{\eta}$ on values. More unusually, a row ρ is interpreted as a protocol list $\mathcal{R}[\![\rho]\!]^{\delta}_{\eta}$, defined as $\bigcup_{\sigma\in\rho} \mathcal{S}[\![\sigma]\!]^{\delta}_{\eta}$, the list concatenation of the interpretations of the elements of ρ. The environment δ maps effect names to effect labels; η maps type variables to semantic types and row variables to protocol lists.

This said, our interpretation of types (Figure 7) is mostly standard [19]. The interpretation of a function type, $\mathcal{V}[\![\tau \xrightarrow{\rho} \kappa]\!]^{\delta}_{\eta}$, is the set of values v such that the application of v to a value w in $\mathcal{V}[\![\tau]\!]^{\delta}_{\eta}$ satisfies a *wp* assertion with protocol list $\mathcal{R}[\![\rho]\!]^{\delta}_{\eta}$ and postcondition $\mathcal{V}[\![\kappa]\!]^{\delta}_{\eta}$. What is crucial is that the validity-and-distinctness property that we have built into the definition of *wp* formalizes the requirement that effect names be pairwise distinct. The interpretation of an effect-polymorphic type involves a quantification $\forall E$ over protocol lists.

Theorem 2 (Fundamental Theorem). *The syntactic judgment entails the semantic judgment: $\varXi \mid \varDelta \mid \varGamma \vdash e\ :\ \rho\ :\ \tau \implies \varXi \mid \varDelta \mid \varGamma \vDash e\ :\ \rho\ :\ \tau$.*

We establish this theorem by induction on the syntactic typing judgment. For every syntactic typing rule, we prove that the interpretation of the conclusion follows from the interpretations of the premises.

The previous two theorems lead directly to the desired type soundness result:

Theorem 3 (Soundness of Tes). *If* $\emptyset \mid \emptyset \mid \emptyset \vdash e : \langle\rangle : \texttt{unit}$*, then* e *is safe.*

6 Related Work

Hillerström and Lindley [14] study the core calculus of Links [9], a functional programming language for web applications, which they extend with support for effect handlers. Taking advantage of Links's row-based approach to type-checking records, they annotate function types with rows of effects. Their rows use Rémy's kind discipline [32] to ensure that an effect name can never appear twice in a row.

Leijen [22] formalizes a subset of the Koka language [23]. He presents a calculus with support for handlers and globally defined effects, a type system with value and effect polymorphism, and a compilation strategy for explicitly-typed programs. This strategy relies on a *selective CPS transformation* [26], which he extends with support for effect polymorphism. A row in Leijen's system is *univariate*: it contains at most one row variable. Tes, in contrast, allows a row to contain several row variables. This ability is exploited, for example, in the typing rule LexHandle. Indeed, the premise contains the effect-polymorphic type $\forall\theta.\ (\alpha \xrightarrow{\theta} \beta) \xrightarrow{\theta\cdot\rho} \tau$, where θ abstracts away the fresh effect label that is allocated by `lex-handle`.

A notable omission from Leijen's formalization is Koka's `inject` [21], which is akin to a `lift` coercion. Biernacki et al. [4] are the first authors to provide a formal treatment of such a construct. They define its operational semantics and they propose a type system with effect polymorphism and univariate rows. They present the first binary logical relations for effect handlers, and they use these relations to prove that their system is sound. In a later paper [5], the same authors introduce λ^{HEL}, a calculus that supports both dynamic allocation of effect labels and effect coercions. In addition to the `lift` coercion, they consider (1) the *swap* coercion, which exchanges two effects in a row; (2) the *cons* coercion, which rearranges effects deep in a row; and (3) composition of coercions. These new coercions do not add expressiveness: they can be expressed in terms of `lift`. Still, they help programmers control the dynamic search for a handler. Biernacki et al. propose a type system with support for universal and existential types. Although `counter`, discussed in Sections 2 and 4, is expressible in λ^{HEL}, Biernacki et al.'s type system does not accept this program. (This has been confirmed by the authors in a personal communication.) The technical reason why `counter` is ill-typed is that the subsumption rules are not sufficiently flexible: an abstract row θ cannot be weakened to a larger row. It is not trivial how to overcome this issue, because the interpretation of a signature in Biernacki et al.'s system depends on the signature's position in the row. Tes, in contrast, allows extension, thanks to the rule Extend.

Zhang and Myers [41] present "a new semantics based on tunneling", which they claim avoids "accidental handling" by construction. As far as we understand, however, they do not propose a semantics in the usual sense, that is, a reduction semantics. Instead, their "semantics" seems to be a translation of the surface language into a core calculus, $\lambda_{\Downarrow\Uparrow}$. This translation is not formally defined: it is sketched by way of examples. Furthermore, as noted by Biernacki et al. [6], there is a discrepancy between the paper presentation of $\lambda_{\Downarrow\Uparrow}$ and its Coq formalization. The paper does not mention dynamic generation of effect labels, but the calculus that is formalized in Coq supports this feature via a construct that generates a fresh effect label and installs a handler for this label; in other words, a lexically scoped handler.

For this calculus with lexically scoped handlers, Zhang and Myers propose a type system with support for effect polymorphism. They prove its soundness using binary logical relations. Then, they exploit these logical relations to establish interesting typed contextual equivalence laws. One law [41, Example 1] shows that an effect-polymorphic function cannot intercept the effects represented by an abstract row variable. This law seems to express the intuitive idea of "absence of accidental handling", but we remark that this notion is never formally defined.

Zhang and Myers [41] and other authors [8] suggest that "absence of accidental handling", sometimes also referred to as "effect safety", has something to do with parametricity. Unfortunately, "parametricity" itself is a somewhat loosely-defined concept. As far as we understand, the word "parametricity" refers to the fact that a syntactic universal type is interpreted via a meta-level universal quantification over a certain universe of semantic types. However, the strength of this meta-level quantification depends on which universe of semantic types is chosen. A smaller universe yields a system with weaker universal types, which may enjoy fewer equivalence laws, but may also admit more well-typed programs.

To illustrate this point, let us ask whether our calculus, TesLang, can be extended with a "dynamic-wind" construct [11]. This construct, `dynamic-wind` $p\ e\ q$, monitors the execution of e and invokes the thunk p whenever control enters e (at the beginning of e's execution and every time e is resumed) and invokes the thunk q whenever control leaves e (at the end of e's execution and every time e performs an effect). To type-check this construct, one might extend Tes with the following typing rule:

$$\textsc{DynamicWind}\quad \frac{\begin{array}{c}\Xi \mid \Delta \mid \Gamma \vdash e : \rho : \tau \\ \Xi \mid \Delta \mid \Gamma \vdash p : \rho : \texttt{unit} \to \texttt{unit} \qquad \Xi \mid \Delta \mid \Gamma \vdash q : \rho : \texttt{unit} \to \texttt{unit}\end{array}}{\Xi \mid \Delta \mid \Gamma \vdash \texttt{dynamic-wind}\ p\ e\ q : \rho : \tau}$$

We have proved that this rule is sound with respect to the interpretation of types presented in Section 5. So, our semantic model supports `dynamic-wind`. Furthermore, our semantic model arguably enjoys "parametricity", since a universal type is interpreted via a meta-level universal quantification. Yet, introducing `dynamic-wind` breaks Zhang and Myers's desired equivalence law [41, Example 1], because it allows observing arbitrary effects, without knowledge of their name and type. Therefore, "parametricity" does not guarantee "absence of accidental handling".

The lesson that we draw from this remark is that a programming language designer is faced with a tension between making the language more powerful by introducing constructs such as `dynamic-wind`, allowing new programs to be written, and making the language less powerful by forbidding such constructs, thereby validating new equivalence laws. Our (unary) semantic model (§5) errs on the side of admitting more constructs and fewer equivalence laws. In future work, it would be interesting to propose a (binary) semantic model that admits fewer constructs and validates more laws, so as to prove that TES without `dynamic-wind` validates Zhang and Myers's law [41, Example 1].

Despite their previous studies of coercions [4,5], Biernacki et al. [6] argue against coercions, which they deem impractical for real-world programming, and propose a type system for a language that supports lexically scoped handlers only. They present two semantics for this language: (1) an *open semantics*, where effect names are not substituted with labels, and where evaluation is defined among open terms in a capture-avoiding way; and (2) a *generative semantics*, where effect names are substituted at runtime with effect labels, as in TESLANG. By means of binary logical relations, they prove that the type system is sound and that the two semantics are equivalent.

Kammar and Pretnar [18] show that a calculus with effects and handlers but without references and without dynamic allocation of effect labels admits a type system with unrestricted polymorphism. Thus, generalization applies even to an expression that performs and handles effects. Kammar and Pretnar establish the soundness of their system via a syntactic approach [40]. The version of TES that we have formalized in Coq [36] distinguishes *pure* and *impure* expressions and allows generalizing the type of a pure expression. The pure expressions include expressions that perform or handle effects. Allocating a fresh effect label is still considered impure. Although such an allocation seems intuitively harmless, our current semantic model interprets allocation as an Iris "update", and Iris does not allow exchanging a universal quantifier with an update modality, so we are unable to justify that allocation is pure. We conjecture that this problem would perhaps not appear in a syntactic approach.

7 Conclusion

In this paper, we have argued in favor of a simple semantics for effect handlers, where the dynamic search for a handler is based purely on equality of effect labels, and where fresh labels can be generated at runtime. This language can express, but is not restricted to, lexically scoped handlers. We have proposed a type system equipped with type and effect polymorphism and with a powerful subsumption relation. A distinguishing feature is the idea that a row expresses a disjointness requirement on effect labels. We have established type soundness via a semantic approach.

In future work, it would be desirable to strengthen our semantic model and turn it into a binary model, so as to establish contextual equivalence laws such as Zhang and Myers's [41]. We also wish to investigate support for modules and inference of principal types, with the ultimate aim of proposing a strong type system for OCaml 5.

References

1. Ahmed, A.J., Fluet, M., Morrisett, G.: A step-indexed model of substructural state. In: International Conference on Functional Programming (ICFP). pp. 78–91 (Sep 2005)
2. Bauer, A., Pretnar, M.: An effect system for algebraic effects and handlers. Logical Methods in Computer Science **10**(4) (2014)
3. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. Journal of Logical and Algebraic Methods in Programming **84**(1), 108–123 (2015)
4. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Handle with care: relational interpretation of algebraic effects and handlers. Proceedings of the ACM on Programming Languages **2**(POPL), 8:1–8:30 (2018)
5. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Abstracting algebraic effects. Proceedings of the ACM on Programming Languages **3**(POPL), 6:1–6:28 (2019)
6. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Binders by day, labels by night: effect instances via lexically scoped handlers. Proceedings of the ACM on Programming Languages **4**(POPL), 48:1–48:29 (2020)
7. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effects as capabilities: effect handlers and lightweight effect polymorphism. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 126:1–126:30 (2020)
8. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. Journal of Functional Programming **30**, e8 (2020)
9. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: Formal Methods for Components and Objects. Lecture Notes in Computer Science, vol. 4709, pp. 266–296. Springer (Nov 2006)
10. Dolan, S., White, L.: Syntax with shifted names (Aug 2019), presented at the Workshop on Type-driven Development (TyDe)
11. Flatt, M., Yu, G., Findler, R.B., Felleisen, M.: Adding delimited and composable control to a production programming environment. In: International Conference on Functional Programming (ICFP). pp. 165–176 (Oct 2007)
12. Garrigue, J.: Relaxing the value restriction. In: Functional and Logic Programming. Lecture Notes in Computer Science, vol. 2998, pp. 196–213. Springer (Apr 2004)
13. Hendriks, D., van Oostrom, V.: adbmal. In: International Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 2741, pp. 136–150. Springer (Aug 2003)
14. Hillerström, D., Lindley, S.: Liberating effects with rows and handlers. In: International Workshop on Type-Driven Development (TyDe@ICFP). pp. 15–27 (Sep 2016)
15. Hillerström, D., Lindley, S.: Shallow effect handlers. In: Asian Symposium on Programming Languages and Systems (APLAS). Lecture Notes in Computer Science, vol. 11275, pp. 415–435. Springer (Dec 2018)
16. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming **28**, e20 (2018)
17. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: International Conference on Functional Programming (ICFP). pp. 145–158 (Sep 2013)
18. Kammar, O., Pretnar, M.: No value restriction is needed for algebraic effects and handlers. Journal of Functional Programming **27**, e7 (2017)

19. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: Principles of Programming Languages (POPL) (Jan 2017)
20. Krogh-Jespersen, M., Svendsen, K., Birkedal, L.: A relational model of type-and-effects in higher-order concurrent separation logic. In: Principles of Programming Languages (POPL). pp. 218–231 (Jan 2017)
21. Leijen, D.: Koka: Programming with row polymorphic effect types. In: Workshop on Mathematically Structured Functional Programming (MSFP). vol. 153, pp. 100–126 (Apr 2014)
22. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: Principles of Programming Languages (POPL). pp. 486–499 (Jan 2017)
23. Leijen, D.: Koka. `https://www.microsoft.com/en-us/research/project/koka/` (2020)
24. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system: documentation and user's manual (Sep 2019)
25. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML – Revised. MIT Press (May 1997)
26. Nielsen, L.R.: A selective CPS transformation. Electronic Notes in Theoretical Computer Science **45**, 311–331 (Nov 2001)
27. Odersky, M., Boruch-Gruszecki, A., Brachthäuser, J.I., Lee, E., Lhoták, O.: Safer exceptions for Scala. In: Symposium on Scala. pp. 1–11 (Oct 2021)
28. Pessaux, F., Leroy, X.: Type-based analysis of uncaught exceptions. ACM Transactions on Programming Languages and Systems **22**(2), 340–377 (2000)
29. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
30. Plotkin, G.D., Pretnar, M.: Handlers of algebraic effects. In: European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 5502, pp. 80–94. Springer (Mar 2009)
31. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science (LICS). pp. 55–74 (2002)
32. Rémy, D.: Type checking records and variants in a natural extension of ML. In: Principles of Programming Languages (POPL). pp. 77–88 (1989)
33. The Iris Team: HeapLang. `https://gitlab.mpi-sws.org/iris/iris/-/blob/master/iris_heap_lang/lang.v` (2022)
34. Tofte, M.: Type inference for polymorphic references. Information and Computation **89**(1), 1–34 (1990)
35. de Vilhena, P.E., Pottier, F.: A separation logic for effect handlers. Proceedings of the ACM on Programming Languages **5**(POPL) (Jan 2021)
36. de Vilhena, P.E., Pottier, F.: A type system for effect handlers and dynamic labels: Coq formalization. `https://gitlab.inria.fr/pdevilhe/tes` (2022)
37. Vindum, S.F., Birkedal, L.: Contextual refinement of the Michael-Scott queue. In: Certified Programs and Proofs (CPP). pp. 76–90 (Jan 2021)
38. Wand, M.: Type inference for record concatenation and multiple inheritance. Information and Computation **93**(1), 1–15 (Jul 1991)
39. Wright, A.K.: Simple imperative polymorphism. Lisp and Symbolic Computation **8**(4), 343–356 (Dec 1995)
40. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1), 38–94 (Nov 1994)
41. Zhang, Y., Myers, A.C.: Abstraction-safe effect handlers via tunneling. Proceedings of the ACM on Programming Languages **3**(POPL), 5:1–5:29 (2019)

42. Zhang, Y., Salvaneschi, G., Beightol, Q., Liskov, B., Myers, A.C.: Accepting blame for safe tunneled exceptions. In: Programming Language Design and Implementation (PLDI). pp. 281–295 (Jun 2016)

Interpreting Knowledge-based Programs

Alexander Knapp[1](✉), Heribert Mühlberger[1], and Bernhard Reus[2]

[1] Universität Augsburg, Augsburg, Germany
{knapp, muehlber}@informatik.uni-augsburg.de

[2] University of Sussex, Brighton, UK
bernhard@sussex.ac.uk

Abstract Knowledge-based programs specify multi-agent protocols with epistemic guards that abstract from how agents learn and record facts or information about other agents and the environment mutual dependency between the evaluation of epistemic guards over the reachable states and the derivation of the reachable states depending on the evaluation of epistemic guards synchronous programming languages to the interpretation problem of knowledge-based programs and demonstrate that the resulting constructive interpretation is monotone and has a least fixed point. We relate our approach with existing interpretation schemes for both synchronous and asynchronous programs interpretation and illustrate the procedure by several examples and an application to the Java memory model.

1 Introduction

Knowledge-based programs [14] describe multi-agent systems based on explicit knowledge tests on what an agent knows or does not know about itself, other agents, and the environment: Extending standard programs, an agent may look beyond what it can directly observe by reasoning about the possible states of the other agents and the environment in all possible program executions. Such non-local, epistemic conditions abstract from how an agent may learn and record particular environmental facts or information about other agents. Thus knowledge-based programs rather are specifications of (multi-agent) protocols that may be implemented by standard, directly executable programs. For being implementable in the first place, however, it has to be ensured that the knowledge guards can be resolved consistently given all possible program executions.

Consider for example a bit transmission [14, Ex. 4.1.1, Ex. 7.1.1], where a sender S has to transmit a bit sbit over a lossy channel to a receiver R who has to acknowledge the reception, again over a lossy channel. This can be modelled by a knowledge-based program over the state variables $\mathrm{sbit} \in \{0, 1\}$, $\mathrm{rval} \in \{\bot, 0, 1\}$, and $\mathrm{ack} \in \{0, 1\}$ as follows: S can only directly observe (read) sbit and ack, and R only rval (but both may write all variables); $(\mathsf{K}_{\mathrm{R}}\, \mathrm{sbit} = 0) \lor (\mathsf{K}_{\mathrm{R}}\, \mathrm{sbit} = 1)$ expresses that R knows sbit's value and is abbreviated by $\mathsf{K}_{\mathrm{R}}\, \mathit{sbit}$. The behaviour description consists of a looping guarded command with two branches that is started with $\mathrm{rval} = \bot$ and $\mathrm{ack} = 0$, but sbit left undetermined:

$$\begin{array}{ll} \textbf{do}\ \neg\mathsf{K}_{\mathrm{S}}\, \mathsf{K}_{\mathrm{R}}\, \mathit{sbit} \twoheadrightarrow (\mathrm{rval} \leftarrow \mathrm{sbit}\ \texttt{or}\ \texttt{skip}) & \text{— S} \\ [\!]\ \mathsf{K}_{\mathrm{R}}\, \mathit{sbit} \land \neg\mathsf{K}_{\mathrm{R}}\, \mathsf{K}_{\mathrm{S}}\, \mathsf{K}_{\mathrm{R}}\, \mathit{sbit} \twoheadrightarrow (\mathrm{ack} \leftarrow 1\ \texttt{or}\ \texttt{skip})\ \textbf{od} & \text{— R} \end{array}$$

The guarded branches are separated by a $[\!]$, `or` means a non-deterministic choice, and `skip` doing nothing: S sends the bit as long as it does not know that R received it, and R

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 253–280, 2023.
https://doi.org/10.1007/978-3-031-30044-8_10

keeps acknowledging once it has learnt the bit and does not know that S knows this fact. The epistemic formulæ $\mathsf{K}_a\,\varphi$ in the program are to be interpreted as in classical Kripke semantics: φ holds in all states (or worlds) that agent a currently deems possible. Which states these are is regulated on the one hand by what a can observe: any state that is indistinguishable from the current one by the available observations is possible for the agent. In the example only S can observe sbit, though, due to the protocol, it should be possible that eventually R knows its value. On the other hand, the possible states depend on which runs of the knowledge-based program may actually happen, i.e., which states are reachable taking epistemically guarded transitions: If only the actions of the program are taken, it is impossible to reach a state satisfying both $\mathrm{rval} \neq \bot$ and $\mathrm{rval} \neq \mathrm{sbit}$, which, however, is present in the global state space; but it is decisive that it is not reachable in any execution in order to have some execution where $\mathsf{K}_{\mathrm{R}}\ \mathit{sbit}$ can become true.

The interpretation of knowledge-based programs hinges precisely on this mutual dependency between the evaluation of epistemic guards over the reachable states and the derivation of the reachable states depending on the evaluation of the epistemic guards. This implicit definition of the epistemic state of the agents by the observables and the reachable states of the commonly known protocol is in stark contrast to Baltag's epistemic action models [4,31], where the epistemic state is given and manipulated explicitly. In many cases, including the bit transmission protocol, the reachable state space may be computed using static analysis techniques without taking into account the epistemic nature of the guards. However, the interplay between knowledge and reachability may sometimes become more intricate: The more states are reachable the less is known definitely, and the guards will in turn influence what is reachable positively or negatively.

Consider, for another example, a variable setting problem [14, Exc. 7.5] involving a single agent a and a single state variable $\mathrm{x} \in \{0, 1, 2, 3\}$, where a cannot observe x directly. The agent executes the following guarded command starting with $\mathrm{x} = 0$:

```
if $\mathsf{K}_\mathrm{a}\,\mathrm{x} \neq 1 \rightarrowtriangle \mathrm{x} \leftarrow 3$
  $[\!]$ $\mathsf{K}_\mathrm{a}\,\mathrm{x} \neq 3 \rightarrowtriangle \mathrm{x} \leftarrow 1$ fi
```

Being an initial condition, $\mathrm{x} = 0$ is reachable, whereas $\mathrm{x} = 2$ is not reachable as 2 is never assigned. However, two different sets of reachable states make for a consistent interpretation of the knowledge guards for the remaining values: $\{\mathrm{x} = 0, \mathrm{x} = 1\}$, where $\mathsf{K}_\mathrm{a}\,\mathrm{x} \neq 1$ is false and $\mathsf{K}_\mathrm{a}\,\mathrm{x} \neq 3$ is true, and $\{\mathrm{x} = 0, \mathrm{x} = 3\}$, with the opposite results. The singleton set $\{\mathrm{x} = 0\}$ is ruled out, since both guards would be true such that $\mathrm{x} = 3$ and $\mathrm{x} = 1$ are reachable; and $\{\mathrm{x} = 0, \mathrm{x} = 1, \mathrm{x} = 3\}$ is impossible, since both guards are false and thus neither $\mathrm{x} = 1$ nor $\mathrm{x} = 3$ are reachable. Breaking this cycle by making one of the transitions unconditional on knowledge as, e.g., in

```
if $\mathsf{K}_\mathrm{a}\,\mathrm{x} \neq 1 \rightarrowtriangle \mathrm{x} \leftarrow 3$
  $[\!]$ $\mathsf{K}_\mathrm{a}\,\mathrm{x} \neq 3 \rightarrowtriangle \mathrm{x} \leftarrow 2$
  $[\!]$ $\mathrm{true} \rightarrowtriangle \mathrm{x} \leftarrow 1$        fi
```

yields a knowledge-based program with the unique consistent interpretation $\{\mathrm{x} = 1, \mathrm{x} = 2\}$. For computing its behaviour, however, several steps are needed, first reasoning that $\mathrm{x} = 1$ is reachable, then that $\mathrm{x} = 3$ is not reachable, and, finally, that $\mathrm{x} = 2$ is reachable.

Related Work. In their introduction and seminal treatise on knowledge-based programs [13,14], Fagin et al. characterise the unique interpretability of such programs by their "dependence on the past" w. r. t. some non-empty class of transition systems: The evaluation of knowledge guards in a state coincides for all interpretations in the class that share a common past of the state. A sufficient condition for this dependence is that the program "provides epistemic witnesses" for all interpretations of the class such that not knowing something at some point in time has a counter example in the past. A sufficient condition for this provision, in turn, is that the program is "synchronous", i. e., that all agents can determine the global time from their local states. For example, the bit transmission protocol provides epistemic witnesses and thus is uniquely interpretable; but it is not synchronous. The cycle-breaking variable setting program is also uniquely interpretable, but does not provide epistemic witnesses. For "asynchronous" knowledge-based programs, De Haan et al. [10] suggest to rely on classical iteration of the non-monotone reachability functional that interprets the knowledge modalities according to what currently is assumed to be reachable. The computation process is started with all states assumed to be reachable and stops when some set of states is repeated. This approach fixes some semantics for all knowledge-based programs, also for those which are cyclic and contradictory or only self-fulfilling.

The problem of mutual dependence of guard evaluation and reachability has also occurred in the design of synchronous programming languages [6] for embedded systems, like Esterel [7] or Lustre [18], which rely on "perfect synchrony": a step for reacting to some inputs takes zero time and output signals are produced at exactly the same time as the input signals. Since thus the status of a signal to be produced can be queried at the same time, this requires "logical coherence" saying that a (non-input) signal is present in a step of execution if, and only if, a command emitting this signal is executed in this step. Whereas Lustre forbids cyclic programs on a syntactic basis, Berry's approach to the semantics of Esterel [8] singles out "reactive" — at least one execution — and "determinate" — at most one execution — programs using a static executability analysis: It is computed which signals *must* be present, i. e., have to occur inevitably, and which signals *cannot* be present, i. e., have no emitting execution. This is also referred to as must/cannot analysis and has to be performed several times for finding a fixed point of all the signal statuses.

In logic programming involving "negation as failure" under- and over-approximations in terms of three- and four-valued logics lead to the "Kripke-Kleene fixpoint" and "well-founded" models; see [11] for an overview. There, however, the temporal dimension of reachability or executability is not involved. The "stable model semantics" [16,5] stresses the rational inclusion or exclusion of atoms: A set of atoms M is "stable" for a logic program Π if it coincides with the minimal set of atoms inferable from the "reduct" Π_M which is obtained from Π by deleting each clause that has a negative literal $\neg p$ in its body with $p \in M$, and all negative literals in the bodies of the remaining clauses. The definition is not algorithmic or constructive; the minimality condition rules out self-fulfilling solutions, the reduction process avoids contradictions. Gelfond's "epistemic specifications" [15] extend (disjunctive) logic programs with a modality K for "subjective literals" for representing incomplete information in programs with several stable models.

Contributions. We apply the principles of the must/cannot analysis to the interpretability problem of knowledge-based programs. After recalling some basic notions of epistemic logic and epistemic transition structures (Sect. 2), we first recapitulate the approaches by Fagin et al. [14] and De Haan et al. [10] in terms of epistemically guarded transition systems, a syntax-agnostic format for knowledge-based programs (Sect. 3). For a more direct analysis, our account of those designs is state-based rather than run-based. We demonstrate the results and the limits of both interpretation schemes by several examples that illustrate (a-)synchronicity and non-monotone interpretation for cyclic, contradictory, or self-fulfilling programs. The latter behaviour is the main motivation for our reformulation of the interpretation problem in terms of epistemic must/can transition structures which offer lower and upper bounds on the behaviour of a knowledge-based program (Sect. 4). We show that this constructive interpretation is always monotone and yields a least fixed point. However, lower and upper bound of the fixed point need not always coincide and we relate decided fixed points with the notions of "providing epistemic witnesses" and synchronicity. We then derive a representation of the behaviour of a knowledge-based program as a general rule system with not only positive but also negative premisses (Sect. 5). Such rule systems correspond to logic programs involving "negation as failure" and the intended solutions form "stable models". The must/can approximation technique, its monotonicity, and it fixed point properties directly transfer to such rule systems. We finally describe an implementation of our constructive interpretation approach in the "Temporal Epistemic Model Interpreter and Checker" (tEmIc, Sect. 6). For model checking interpreted knowledge-based programs, the tool supports CTLK, the combination of "Computational Tree Logic" (CTL) with epistemic logic. Moreover, this logic can also be used in program guards; the interpretation of such temporal-epistemic programs extends the previous approaches. We give some applications to the analysis of the Java memory model.

2 Epistemic Logic and Epistemic Transition Structures

We briefly summarise the basic notions of epistemic logic for expressing knowledge guards [31,30]. We then define epistemic transition structures as the domain of interpretation of knowledge-based programs. These transition structures combine the temporal dimension of executing a program with the epistemic dimension for evaluating what agents know. Both the logic and the transition structures are built over an *epistemic signature* $\Sigma = (P, A)$ that consists of a set of *propositions* P and a set of *agents* A.

2.1 Epistemic Logic

An *epistemic structure* $K = (W, R, L)$ over (P, A) is given by a set of *worlds* W, an A-family of epistemic *accessibility relations* $R = (R_a \subseteq W \times W)_{a \in A}$, and a *labelling* $L \colon W \to \wp P$ assigning each world a set of propositions. In concrete examples, we will require R_a to be an equivalence relation such that if $(w_1, w_2) \in R_a$, then agent a cannot distinguish between the two worlds w_1 and w_2. The *epistemic formulæ* $\varphi \in \Phi_{P,A}$ over (P, A) are defined by the following grammar:

$$\varphi ::= p \mid \text{false} \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{K}_a\, \varphi$$

where $p \in P$ and $a \in A$. The epistemic formula $\mathsf{K}_a\,\varphi$ is to be read as "agent a *knows* φ". We use the usual propositional abbreviations true for ¬false and $\varphi_1 \vee \varphi_2$ for $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$. Furthermore, we consider the epistemic modality M as the dual of K, such that $\mathsf{M}_a\,\varphi$ abbreviates $\neg\mathsf{K}_a\,\neg\varphi$ and is to be read as "agent a *deems* φ *possible*". The *satisfaction relation* of an epistemic formula $\varphi \in \Phi_{P,A}$ over an epistemic structure $K = (W, R, L)$ over (P, A) at a world $w \in W$, written $K, w \models \varphi$, is inductively defined by

$$\begin{aligned}
&K, w \models p \iff p \in L(w)\\
&K, w \not\models \text{false}\\
&K, w \models \neg\varphi \iff K, w \not\models \varphi\\
&K, w \models \varphi_1 \wedge \varphi_2 \iff K, w \models \varphi_1 \text{ and } K, w \models \varphi_2\\
&K, w \models \mathsf{K}_a\,\varphi \iff K, w' \models \varphi \text{ f. a. } w' \in W \text{ with } (w, w') \in R_a
\end{aligned}$$

2.2 Epistemic Transition Structures

An epistemic transition structure combines a temporal transition relation with an epistemic accessibility relation over a common set of states. The transitions describe which states can be reached from a set of initial states, the accessibilities specify which states are indistinguishable. Knowledge formulæ are evaluated over the associated global epistemic structure. This derived structure has the reachable states as its worlds and reuses the accessibility relation and the labelling but restricted to the reachable states.

Formally, an *epistemic transition structure* $M = (S, E, L, S_0, T)$ over (P, A) is given by an epistemic structure (S, E, L), a set of temporally *initial states* $S_0 \subseteq S$, and a temporal *transition relation* $T \subseteq S \times S$. We write $S(M)$ for S, $T(M)$ for T, etc. The (temporally) *reachable states* $S_\omega(M) = \bigcup_{0 \leq k} S_k(M)$ and *transition relation* $T_\omega(M) = \bigcup_{0 \leq k} T_k(M)$ of M are inductively defined by

$$\begin{aligned}
&S_0(M) = S_0, \quad S_{k+1}(M) = S_k(M) \cup \{s' \mid \text{ex. } s \in S_k(M) \text{ s. t. } (s, s') \in T\}\ ;\\
&T_0(M) = \emptyset, \quad T_{k+1}(M) = T_k(M) \cup \{(s, s') \in T \mid s \in S_k(M)\}\ .
\end{aligned}$$

The associated *epistemic structure* of M is given by

$$K(M) = (S_\omega(M), E \cap S_\omega(M)^2, L{\restriction}S_\omega(M))$$

where $S_\omega(M)^2$ abbreviates $S_\omega(M) \times S_\omega(M)$ and $L{\restriction}S_\omega(M)$ denotes labelling L restricted to domain $S_\omega(M)$. The *satisfaction relation* of an epistemic formula $\varphi \in \Phi_{P,A}$ over M at an $s \in S_\omega(M)$, written $M, s \models \varphi$, is defined as

$$M, s \models \varphi \iff K(M), s \models \varphi\ .$$

The set of epistemic transition structures over $\Sigma = (P, A)$ sharing the same *epistemic state basis* $\mathsf{B} = (S, E, L, S_0)$ is denoted by $\mathscr{M}_\Sigma(\mathsf{B})$. We say that $M_1 \subseteq M_2$ for $M_1, M_2 \in \mathscr{M}_\Sigma(\mathsf{B})$ if $T(M_1) \subseteq T(M_2)$ and similarly extend union and intersection from transition relations to epistemic transition structures.

3 Knowledge-based Programs

Knowledge-based programs extend standard programs by explicit knowledge tests. Their interpretation involves a cycle: the evaluation of the epistemic guards depends on the program's reachable states, the derivation of the reachable states on the evaluation of the program's epistemic guards.

We render knowledge-based programs in a syntax-agnostic format as epistemically guarded transition systems. Like epistemic transition structures, these guarded systems operate on a global set of states with epistemic accessibilities and a propositional labelling. All program steps are represented as knowledge-guarded actions of the form $\varphi \supset B$ with φ an epistemic formula and B a relation on the semantic states. Knowledge-independent decisions are obtained by choosing $\varphi = \mathrm{true}$, and any kind of program control structure can be expressed by a judicious choice of guarded actions.

Breaking up the cyclic step of assigning meaning to a knowledge-based program, an epistemically guarded transition system Γ is interpreted over an epistemic transition structure M yielding another epistemic transition structure Γ^M. A guarded action $\varphi \supset B$ of Γ contributes those $(s, s') \in B$ for which $M, s \models \varphi$, where, in particular, s is reachable in M. What is sought for is a consistent interpretation with $\Gamma^M = M$ such that reachability and knowledge are mutually justified. Finding such a balanced structure is complicated by the fact that the interpretation functional is not monotone in general: The more is reachable the less is known and this may make more or less states reachable.

After introducing and illustrating our format of knowledge-based programs we summarise and adapt two existing approaches to their interpretation that have been devised for run-based rather than state-based systems: De Haan et al. [10] propose to iterate the interpretation functional starting from an epistemic transition structure where all states are reachable. Iteration stops when either a fixed point is reached or, due to non-monotonicity, a contradiction is found. In this way all knowledge-based programs are assigned some semantics and there is no distinction between meaningful and contradictory or just self-fulfilling programs. The original approach by Fagin et al. [13,14] characterises knowledge-based programs that admit a unique consistent interpretation by the notion of dependence on the past. A sufficient condition of providing epistemic witnesses is developed which, in particular, applies to the subclass of synchronous knowledge-based programs.

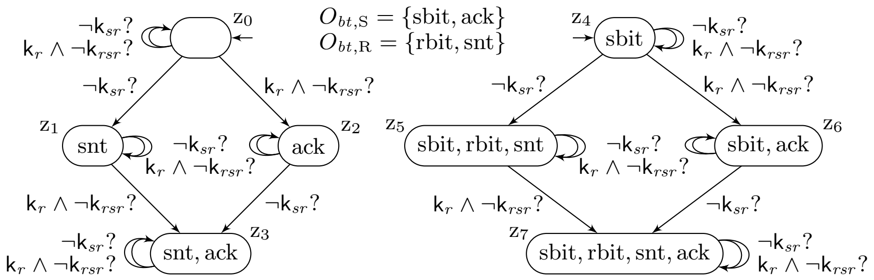
3.1 Epistemically Guarded Transition Systems

An *epistemically guarded transition system* $\Gamma = (S, E, L, S_0, \mathcal{T})$ over (P, A) is given by an epistemic state basis (S, E, L, S_0) over (P, A) and a set $\mathcal{T}$ of *epistemically guarded actions* $\varphi \supset B$ consisting of an epistemic formula $\varphi \in \Phi_{P,A}$ as *guard* and a transition relation $B \subseteq S \times S$.

Example 1. (a) Consider the bit transmission problem of the introduction:

```
do $\neg \mathsf{K}_\mathrm{S}\, \mathsf{K}_\mathrm{R}\, \mathit{sbit} \twoheadrightarrow (\mathrm{rval} \leftarrow \mathrm{sbit}$ or skip$)$
 [] $\mathsf{K}_\mathrm{R}\, \mathit{sbit} \wedge \neg \mathsf{K}_\mathrm{R}\, \mathsf{K}_\mathrm{S}\, \mathsf{K}_\mathrm{R}\, \mathit{sbit} \twoheadrightarrow (\mathrm{ack} \leftarrow 1$ or skip$)$ od
```

A sender agent S sends a bit $\mathrm{sbit} \in \{0, 1\}$ to a receiver agent R over an unreliable channel by setting $\mathrm{rval} \in \{\bot, 0, 1\}$; and R acknowledges the reception over an unreliable channel by setting $\mathrm{ack} \in \{0, 1\}$. Again, we abbreviate $(\mathsf{K}_\mathrm{R}\, \neg\mathrm{sbit}) \vee (\mathsf{K}_\mathrm{R}\, \mathrm{sbit})$ expressing that the receiver knows the bit to be sent by $\mathsf{K}_{\mathrm{R}}\ \mathit{sbit}$. We concretise the problem into an epistemically guarded transition system $\mathit{\Gamma}_{bt} = (\mathsf{B}_{bt}, \mathcal{T}_{bt})$ with $\mathsf{B}_{bt} = (S_{bt}, E_{bt}, L_{bt}, S_{bt,0})$ over $\mathit{\Sigma}_{bt} = (P_{bt}, A_{bt})$ with $P_{bt} = \{\mathrm{sbit}, \mathrm{rbit}, \mathrm{snt}, \mathrm{ack}\}$ and $A_{bt} = \{\mathrm{S}, \mathrm{R}\}$. Since we use a propositional encoding, we represent $\mathrm{rval} \in \{\bot, 0, 1\}$ by a proposition rbit for the transmitted bit and a proposition snt for the validity of rbit. Further abbreviating the knowledge guards $\mathsf{K}_{\mathrm{R}}\ \mathit{sbit}$ by k_r, $\mathsf{K}_{\mathrm{S}}\ \mathsf{K}_{\mathrm{R}}\ \mathit{sbit}$ by k_{sr}, and $\mathsf{K}_{\mathrm{R}}\ \mathsf{K}_{\mathrm{S}}\ \mathsf{K}_{\mathrm{R}}\ \mathit{sbit}$ by k_{rsr}, the transition system $\mathit{\Gamma}_{bt}$ is graphically given by



The states S_{bt} comprise of $\{\mathrm{z}_0, \mathrm{z}_1, \dots, \mathrm{z}_7\}$ with $L_{bt}(\mathrm{z}_0) = \emptyset$, $L_{bt}(\mathrm{z}_1) = \{\mathrm{snt}\}, \dots,$ $L_{bt}(\mathrm{z}_7) = \{\mathrm{sbit}, \mathrm{rbit}, \mathrm{snt}, \mathrm{ack}\}$ as outlined in the graph above; the set of initial states is $S_{bt,0} = \{\mathrm{z}_0, \mathrm{z}_4\}$. The epistemic accessibility relations $E_{bt,a}$ for $a \in A_{bt}$ are given by ***observability sets*** $O_{bt,a}$ that declare two states $s_1, s_2 \in S_{bt}$ to be $O_{bt,a}$*-indistinguishable*, written as $s_1 \sim_{O_{bt,a}} s_2$, if for all $p \in O_{bt,a}$ it holds that $p \in L_{bt}(s_1) \iff p \in L_{bt}(s_2)$, and consequently $E_{bt,a} = \sim_{O_{bt,a}}$, such that $E_{bt,a}$ forms an equivalence relation. Due to $\mathrm{sbit} \notin O_{bt,\mathrm{R}}$, the receiver R cannot "see" sbit and hence cannot distinguish between states z_0 and z_4, but S can. On the other hand, R can distinguish between z_1 and z_5 as R has access to rbit. Finally, $\mathcal{T}_{bt}$ consists of two epistemically guarded actions

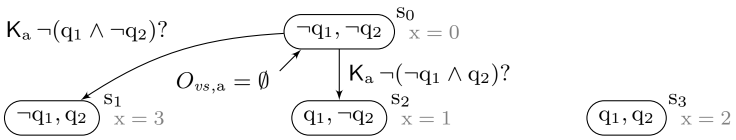
$$\begin{aligned}
&\neg\mathsf{K}_{\mathrm{S}}\ \mathsf{K}_{\mathrm{R}}\ \mathit{sbit} \supset \{(\mathrm{z}_i, \mathrm{z}_i) \mid 0 \le i \le 7\} \cup \{(\mathrm{z}_0, \mathrm{z}_1), (\mathrm{z}_2, \mathrm{z}_3), (\mathrm{z}_4, \mathrm{z}_5), (\mathrm{z}_6, \mathrm{z}_7)\} \ \text{ and} \\
&\mathsf{K}_{\mathrm{R}}\ \mathit{sbit} \wedge \neg\mathsf{K}_{\mathrm{R}}\ \mathsf{K}_{\mathrm{S}}\ \mathsf{K}_{\mathrm{R}}\ \mathit{sbit} \supset \{(\mathrm{z}_i, \mathrm{z}_i) \mid 0 \le i \le 7\} \cup \\
&\qquad\qquad\qquad\qquad \{(\mathrm{z}_0, \mathrm{z}_2), (\mathrm{z}_1, \mathrm{z}_3), (\mathrm{z}_4, \mathrm{z}_6), (\mathrm{z}_5, \mathrm{z}_7)\}\ ,
\end{aligned}$$

which directly reflect the sending and acknowledging actions of the bit transmission problem: The system can only advance from z_0 to z_1 (and z_4 to z_5), where sending has been done successfully, if S does not know that R knows the bit; but it need not make such progress, i.e., sending can be unsuccessful. Similarly, the system can only advance from z_1 to z_3 (and z_5 to z_7), where an acknowledgement has been sent successfully, if R knows the bit and R does not know that S knows that R knows the bit.

(b) Consider the variable setting problem of the introduction for a single agent a:

```
if $\mathsf{K}_{\mathrm{a}}\, \mathrm{x} \neq 1 \rightarrow \mathrm{x} \leftarrow 3$
 [] $\mathsf{K}_{\mathrm{a}}\, \mathrm{x} \neq 3 \rightarrow \mathrm{x} \leftarrow 1$ fi
```

Encoding the integer $x \in \{0, 1, 2, 3\}$ by two bits q_1 and q_2, we model the problem as the following epistemically guarded transition system $\mathit{\Gamma}_{vs} = (\mathsf{B}_{vs}, \mathcal{T}_{vs})$ with $\mathsf{B}_{vs} = (S_{vs},$ $E_{vs}, L_{vs}, S_{vs,0})$ over $\mathit{\Sigma}_{vs} = (P_{vs}, A_{vs})$ with $P_{vs} = \{\mathrm{q}_1, \mathrm{q}_2\}$ and $A_{vs} = \{\mathrm{a}\}$:



$\mathsf{K}_{\mathrm{a}}\,\neg(\mathrm{q}_1 \wedge \neg\mathrm{q}_2)?$
$\neg\mathrm{q}_1, \neg\mathrm{q}_2$
s_0
$\mathrm{x} = 0$
$O_{vs,\mathrm{a}} = \emptyset$
$\mathsf{K}_{\mathrm{a}}\,\neg(\neg\mathrm{q}_1 \wedge \mathrm{q}_2)?$
$\neg\mathrm{q}_1, \mathrm{q}_2$
s_1
$\mathrm{x} = 3$
$\mathrm{q}_1, \neg\mathrm{q}_2$
s_2
$\mathrm{x} = 1$
$\mathrm{q}_1, \mathrm{q}_2$
s_3
$\mathrm{x} = 2$

$O_{vs,\mathrm{a}}$ represents a "blind" agent a that deems all states equally accessible. State s_3 is definitely not reachable. $\mathcal{T}_{vs}$ consists of the epistemically guarded actions

$$\mathsf{K}_{\mathrm{a}}\,\neg(\mathrm{q}_1 \wedge \neg\mathrm{q}_2) \supset \{(\mathrm{s}_0, \mathrm{s}_1)\} \quad \text{and} \quad \mathsf{K}_{\mathrm{a}}\,\neg(\neg\mathrm{q}_1 \wedge \mathrm{q}_2) \supset \{(\mathrm{s}_0, \mathrm{s}_2)\}\ . \qquad \square$$

3.2 Interpreting Epistemically Guarded Transition Systems

An epistemically guarded transition system $\Gamma = (S, E, L, S_0, \mathcal{T})$ over (P, A) is *interpreted* over an epistemic transition structure $M \in \mathscr{M}_{P,A}(S, E, L, S_0)$ by interpreting each guarded action $(\varphi \supset B) \in \mathcal{T}$ w.r.t. M as

$$(\varphi \supset B)^M = \{(s, s') \in B \mid s \in S_\omega(M) \text{ and } M, s \models \varphi\}\ ,$$

and combining these interpretations into the epistemic transition structure

$$\Gamma^M = (S, E, L, S_0, \textstyle\bigcup_{\tau \in \mathcal{T}} \tau^M)\ .$$

We call M a *solution* for Γ if $\Gamma^M = M$.

Example 2. For the bit transmission problem as described in Ex. 1(a), the epistemic transition structure $M_{bt} = (\mathsf{B}_{bt}, T_{bt})$ with $T_{bt} = \{(\mathrm{z}_i, \mathrm{z}_i) \mid i \in \{0, 1, 3, 4, 5, 7\}\} \cup \{(\mathrm{z}_0, \mathrm{z}_1), (\mathrm{z}_1, \mathrm{z}_3), (\mathrm{z}_4, \mathrm{z}_5), (\mathrm{z}_5, \mathrm{z}_7)\}$ satisfies ${\Gamma_{bt}}^{M_{bt}} = M_{bt}$. This structure just omits the states z_2 and z_6 with $L_{bt}(\mathrm{z}_2) = \{\mathrm{ack}\}$ and $L_{bt}(\mathrm{z}_6) = \{\mathrm{sbit}, \mathrm{ack}\}$ which are definitely not reachable, as $\mathsf{K}_{\mathrm{R}}\ \mathit{sbit}$ is false in $\mathrm{z}_0 \sim_{O_{bt,\mathrm{R}}} \mathrm{z}_4$. Indeed,

$$\begin{aligned} M_{bt}, s &\models \neg\mathsf{K}_{\mathrm{S}}\,\mathsf{K}_{\mathrm{R}}\ \mathit{sbit} \iff s \in \{\mathrm{z}_0, \mathrm{z}_1, \mathrm{z}_4, \mathrm{z}_5\} \\ M_{bt}, s &\models \mathsf{K}_{\mathrm{R}}\ \mathit{sbit} \iff s \in \{\mathrm{z}_1, \mathrm{z}_3, \mathrm{z}_5, \mathrm{z}_7\} \\ M_{bt}, s &\models \neg\mathsf{K}_{\mathrm{R}}\,\mathsf{K}_{\mathrm{S}}\,\mathsf{K}_{\mathrm{R}}\ \mathit{sbit} \iff s \in \{\mathrm{z}_0, \mathrm{z}_1, \mathrm{z}_3, \mathrm{z}_4, \mathrm{z}_5, \mathrm{z}_7\} \qquad \square \end{aligned}$$

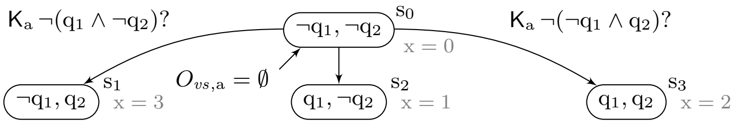
However, finding a solution is complicated by the fact that the functional of interpreting an epistemically guarded transition system over an epistemic transition structure is not monotone, in general, as illustrated by the following examples.

Example 3. (a) Continuing Ex. 1(b) for the variable setting problem Γ_{vs}, consider the epistemic transition structure $M_{vs,0} \in \mathscr{M}_{\Sigma_{vs}}(\mathsf{B}_{vs})$ with the empty transition relation $T(M_{vs,0}) = \emptyset$, and hence $S_0(M_{vs,0}) = \{\mathrm{s}_0\}$. Setting $M_{vs,i+1} = {\Gamma_{vs}}^{M_{vs,i}}$ for $0 \le i \le 2$ we obtain successively

τ	$\tau^{M_{vs,0}}$	$\tau^{M_{vs,1}}$	$\tau^{M_{vs,2}}$
$\mathsf{K}_{\mathrm{a}}\,\neg(\mathrm{q}_1 \wedge \neg\mathrm{q}_2) \supset \{(\mathrm{s}_0, \mathrm{s}_1)\}$	$\{(\mathrm{s}_0, \mathrm{s}_1)\}$	$\emptyset$	$\{(\mathrm{s}_0, \mathrm{s}_1)\}$
$\mathsf{K}_{\mathrm{a}}\,\neg(\neg\mathrm{q}_1 \wedge \mathrm{q}_2) \supset \{(\mathrm{s}_0, \mathrm{s}_2)\}$	$\{(\mathrm{s}_0, \mathrm{s}_2)\}$	$\emptyset$	$\{(\mathrm{s}_0, \mathrm{s}_2)\}$

In particular, $M_{vs,2} = {\mathit{\Gamma}_{vs}}^{M_{vs,1}} = {\mathit{\Gamma}_{vs}}^{{\mathit{\Gamma}_{vs}}^{M_{vs,0}}} = M_{vs,0}$. However, for $M_{vs,4}, M_{vs,5} \in \mathscr{M}_{\Sigma_{vs}}(\mathsf{B}_{vs})$ with $T(M_{vs,4}) = \{(\mathrm{s}_0, \mathrm{s}_1)\}$ and $T(M_{vs,5}) = \{(\mathrm{s}_0, \mathrm{s}_2)\}$ we obtain that ${\mathit{\Gamma}_{vs}}^{M_{vs,4}} = M_{vs,4}$ and ${\mathit{\Gamma}_{vs}}^{M_{vs,5}} = M_{vs,5}$.

(b) For capturing the cycle-breaking variable setting of the introduction consider the following epistemically guarded transition system $\mathit{\Gamma}_{vsb} = (\mathsf{B}_{vs}, \mathcal{T}_{vsb})$ over Σ_{vs} that shares the epistemic state basis B_{vs} with Ex. 1(b):


$\mathsf{K}_\mathrm{a}\,\neg(\mathrm{q}_1 \wedge \neg\mathrm{q}_2)?$
$\neg\mathrm{q}_1, \neg\mathrm{q}_2$
s_0
$\mathrm{x} = 0$
$\mathsf{K}_\mathrm{a}\,\neg(\neg\mathrm{q}_1 \wedge \mathrm{q}_2)?$
$O_{vs,\mathrm{a}} = \emptyset$
$\neg\mathrm{q}_1, \mathrm{q}_2$
s_1
$\mathrm{x} = 3$
$\mathrm{q}_1, \neg\mathrm{q}_2$
s_2
$\mathrm{x} = 1$
$\mathrm{q}_1, \mathrm{q}_2$
s_3
$\mathrm{x} = 2$

For $M_{vsb,0} = (\mathsf{B}_{vs}, \emptyset)$ with $S_0(M_{vsb,0}) = \{\mathrm{s}_0\}$, and setting $M_{vsb,i+1} = {\mathit{\Gamma}_{vsb}}^{M_{vsb,i}}$ for $0 \leq i \leq 3$ we obtain successively

τ	$\tau^{M_{vsb,0}}$	$\tau^{M_{vsb,1}}$	$\tau^{M_{vsb,2}}$	$\tau^{M_{vsb,3}}$
$\mathsf{K}_\mathrm{a}\,\neg(\mathrm{q}_1 \wedge \neg\mathrm{q}_2) \supset \{(\mathrm{s}_0, \mathrm{s}_1)\}$	$\{(\mathrm{s}_0, \mathrm{s}_1)\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\mathrm{true} \supset \{(\mathrm{s}_0, \mathrm{s}_2)\}$	$\{(\mathrm{s}_0, \mathrm{s}_2)\}$	$\{(\mathrm{s}_0, \mathrm{s}_2)\}$	$\{(\mathrm{s}_0, \mathrm{s}_2)\}$	$\{(\mathrm{s}_0, \mathrm{s}_2)\}$
$\mathsf{K}_\mathrm{a}\,\neg(\neg\mathrm{q}_1 \wedge \mathrm{q}_2) \supset \{(\mathrm{s}_0, \mathrm{s}_3)\}$	$\{(\mathrm{s}_0, \mathrm{s}_3)\}$	$\emptyset$	$\{(\mathrm{s}_0, \mathrm{s}_3)\}$	$\{(\mathrm{s}_0, \mathrm{s}_3)\}$

For $M_{vsb,3}$ with $S_\omega(M_{vsb,3}) = \{\mathrm{s}_0, \mathrm{s}_1, \mathrm{s}_3\}$ it finally holds that ${\mathit{\Gamma}_{vsb}}^{M_{vsb,3}} = M_{vsb,3}$. □

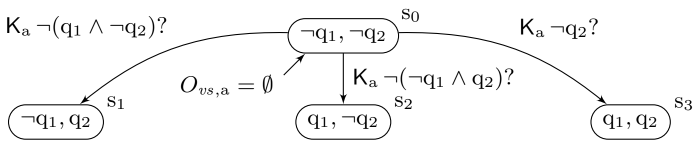
3.3 Iteration Semantics

For illustrating the non-monotonicity of the interpretation functional we have started the interpretation sequence for $\mathit{\Gamma}$ with the smallest epistemic transition structure which suggests to look for a smallest fixed point — which need not exist. De Haan et al. [10] argue that a substitute consisting of the greatest fixed point would be more liberal. They construct a transfinite approximation sequence starting from an N_0 having all states reachable. For a successor ordinal $\alpha+1$, the approximation $N_{\alpha+1}$ is just the interpretation of $\mathit{\Gamma}$ in N_α; for a limit ordinal λ, the approximation $N_\lambda = \bigcap_{\alpha<\lambda} \bigcup_{\alpha\leq\beta<\lambda} N_\beta$ is "the intersection of unions of approximations that are sufficiently close to the limit" [10, p. 269]. The latter is preferred over a union of intersections as it includes more states which implies less knowledge, such that "agents [know] facts only when there are good reasons for them" (ibid.). Due to cardinality reasons, the ordinal $\eta_{\mathit{\Gamma}} = \inf\{\alpha \mid \text{ex. } \beta \text{ s. t. } \alpha < \beta \text{ and } N_\alpha = N_\beta\}$ exists. If $N_{\alpha+1} \subseteq N_\alpha$ for all $\alpha \geq \eta_{\mathit{\Gamma}}$, then $N_{\eta_{\mathit{\Gamma}}+1} = N_{\eta_{\mathit{\Gamma}}}$; otherwise there is some $\alpha \geq \eta_{\mathit{\Gamma}}$ such that $N_{\alpha+1} \not\subseteq N_\alpha$. Thus $\alpha_{\mathit{\Gamma}} = \inf\{\alpha \mid \eta_{\mathit{\Gamma}} \leq \alpha \text{ and } (N_\alpha = N_{\alpha+1} \text{ or } N_{\alpha+1} \not\subseteq N_\alpha)\}$ exists and the *iteration semantics* of $\mathit{\Gamma}$ is defined as $N_{\alpha_{\mathit{\Gamma}}}$. This yields the greatest fixed point if the interpretation functional is monotone.

Example 4. (a) For the variable setting problem $\mathit{\Gamma}_{vs}$ of Ex. 1(b) the interpretation sequence $(N_{vs,\alpha})_{0\leq\alpha}$ starts with $N_{vs,0}$ showing $T(N_{vs,0}) = S_{vs} \times S_{vs}$. Using the epistemic transition structures from Ex. 3(a) it holds that $N_{vs,k+1} = {\mathit{\Gamma}_{vs}}^{N_{vs,k}} = M_{vs,2}$ for k even and $N_{vs,k+1} = M_{vs,1}$ for $k \geq 1$ odd. Thus, $N_{vs,1} = N_{vs,3}$ such that $\eta_{\mathit{\Gamma}_{vs}} = 1 = \alpha_{\mathit{\Gamma}_{vs}}$, since $T(N_{vs,2}) = \{(\mathrm{s}_0, \mathrm{s}_0), (\mathrm{s}_0, \mathrm{s}_1), (\mathrm{s}_0, \mathrm{s}_2)\} \not\subseteq \emptyset = T(N_{vs,1})$. Hence the iteration semantics of $\mathit{\Gamma}_{vs}$ is given by $N_{vs,1} = M_{vs,2}$; since its transition relation is empty, $\mathit{\Gamma}_{vs}$ has the same iteration semantics as an epistemically guarded transition system without any guarded actions.

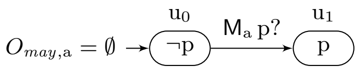
(b) Computing the iteration semantics sequence $(N_{vsb,\alpha})_{0\leq k}$ of the cycle-breaking variable setting $\mathit{\Gamma}_{vsb}$ of Ex. 3(b) proceeds as $N_{vsb,k} = M_{vsb,k+1}$. Since this time the functional is monotone from $\alpha = 1$ onwards, the iteration semantics is $N_{vsb,2}$.

(c) Consider the following epistemically guarded transition system $\mathit{\Gamma}_{nc} = (\mathsf{B}_{vs}, \mathcal{T}_{nc})$ over $\mathit{\Sigma}_{vs}$ that shares the epistemic basis B_{vs} with the variable setting problem $\mathit{\Gamma}_{vs}$ of (a) and only adds the guarded action $\mathsf{K}_{\mathrm{a}}\,\neg \mathrm{q}_2 \supset \{(\mathrm{s}_0, \mathrm{s}_3)\}$:


$\mathsf{K}_{\mathrm{a}}\,\neg(\mathrm{q}_1 \wedge \neg \mathrm{q}_2)?$
s_0
$\neg \mathrm{q}_1, \neg \mathrm{q}_2$
$\mathsf{K}_{\mathrm{a}}\,\neg \mathrm{q}_2?$
$O_{vs,\mathrm{a}} = \emptyset$
$\mathsf{K}_{\mathrm{a}}\,\neg(\neg \mathrm{q}_1 \wedge \mathrm{q}_2)?$
s_1
$\neg \mathrm{q}_1, \mathrm{q}_2$
s_2
$\mathrm{q}_1, \neg \mathrm{q}_2$
s_3
$\mathrm{q}_1, \mathrm{q}_2$

The interpretation process runs as for $\mathit{\Gamma}_{vs}$, and the epistemic transition structure with the empty transition relation is also the iteration semantics of $\mathit{\Gamma}_{nc}$. This time, however, there is a unique non-empty interpretation, viz. the transition structure consisting only of $(\mathrm{s}_0, \mathrm{s}_1)$. Finding this solution is not constructive and some speculation is necessary: there is no solution where s_2 is reachable; if s_2 were reachable, then s_1 would be reachable leading to a contradiction due to the (non-)reachability of s_3. Thus only the possibility of s_0 and s_1 being reachable, and s_2 and s_3 unreachable, remains.

(d) For the epistemically guarded transition system $\mathit{\Gamma}_{may}$ over $(\{\mathrm{p}\}, \{\mathrm{a}\})$ given by


$O_{may,\mathrm{a}} = \emptyset$
u_0
$\neg \mathrm{p}$
$\mathsf{M}_{\mathrm{a}}\,\mathrm{p}?$
u_1
p

the iteration process when started with $N_{may,0}$ having $T(N_{may,0}) = \{\mathrm{u}_0, \mathrm{u}_1\} \times \{\mathrm{u}_0, \mathrm{u}_1\}$ evaluates $\mathsf{M}_{\mathrm{a}}\,\mathrm{p}$ to true and we obtain $N_{may,1}$ with $T(N_{may,1}) = \{(\mathrm{u}_0, \mathrm{u}_1)\}$ which in turn is confirmed by the next iteration yielding a fixed point. This iteration semantics, however, has a touch of a "vaticinium ex eventu": p can be reached since p may be reached. □

3.4 Unique Interpretation Solutions

A knowledge-based program can be executed reliably just step by step if each knowledge guard can be stably decided based on what has been computed up to the current point of execution. In particular, in order to obtain a solution by execution, knowledge must not be invalidated by information only to be gained later on. Conversely, if all knowledge guards can be decided by just looking to the past, there is at most a single solution.

Based on this observation, Fagin et al. [13,14] develop a formal characterisation of unique interpretability by capturing the notion that solutions "depend on the past". They then show that "providing epistemic witnesses" is a sufficient criterion for "dependence on the past", which in turn always holds for "synchronous" programs. We briefly summarise their main line of argument adapting the demonstration from their run-based account for knowledge-based programs to our state-based epistemically guarded transition systems.[3]

[3] The proofs are available in a long version at https://arxiv.org/abs/2301.10807.

An epistemic formula $\varphi \in \Phi_{P,A}$ is said to *depend on the past* w. r. t. a class of epistemic transition structures $\mathcal{M} \subseteq \mathscr{M}_{P,A}(\mathsf{B})$ if for all $M_1, M_2 \in \mathcal{M}$ and all $k \in \mathbb{N}$ it holds that $T_k(M_1) = T_k(M_2)$ implies $M_1, s \models \varphi \iff M_2, s \models \varphi$ for all $s \in S_k(M_1) \cap S_k(M_2)$; an epistemically guarded transition system $\Gamma = (\mathsf{B}, \mathcal{T})$ over (P, A) is *depending on the past* w. r. t. $\mathcal{M}$ if every φ in $(\varphi \supset B) \in \mathcal{T}$ depends on the past w. r. t. $\mathcal{M}$.

Example 5. For Ex. 3(a) neither $\mathsf{K}_\mathrm{a} \neg(\mathrm{q}_1 \wedge \neg \mathrm{q}_2)$ nor $\mathsf{K}_\mathrm{a} \neg(\neg \mathrm{q}_1 \wedge \mathrm{q}_2)$ depends on the past w. r. t. $\{M_{vs,0}, M_{vs,1}\}$. In particular, $T_0(M_{vs,0}) = \emptyset = T_0(M_{vs,1})$ and $S_0(M_{vs,0}) = \{\mathrm{s}_0\} = S_0(M_{vs,1})$, but $M_{vs,0}, \mathrm{s}_0 \models \mathsf{K}_\mathrm{a} \neg(\mathrm{q}_1 \wedge \neg \mathrm{q}_2)$ and $M_{vs,1}, \mathrm{s}_0 \not\models \mathsf{K}_\mathrm{a} \neg(\mathrm{q}_1 \wedge \neg \mathrm{q}_2)$. Similarly for Ex. 3(b), these two formulæ do not depend on the past w. r. t. $\{M_{vsb,0}, M_{vsb,1}, M_{vsb,2}, M_{vsb,3}\}$, but they do w. r. t. $\{M_{vsb,1}, M_{vsb,2}, M_{vsb,3}\}$. □

An epistemically guarded transition system Γ has at most one solution if, and only if, it depends on the past w. r. t. all its solutions. Due to the dependence on the past the successive reachable transition relations $T_k(M)$ of all solutions $M = \Gamma^M$, i. e., their pasts, coincide.

Proposition 1. *Let $\Gamma = (\mathsf{B}, \mathcal{T})$ be an epistemically guarded transition system over Σ. Then Γ has at most one solution if, and only if, there is an $\mathcal{M} \subseteq \mathscr{M}_\Sigma(\mathsf{B})$ with $\{M \in \mathscr{M}_\Sigma(\mathsf{B}) \mid \Gamma^M = M\} \subseteq \mathcal{M}$ such that Γ depends on the past w. r. t. $\mathcal{M}$.*

In order to obtain a solution of Γ by execution, the system is interpreted repeatedly to construct the approximations $(M_k)_{0 \leq k}$ with $M_{k+1} = \Gamma^{M_k}$ for $k \geq -1$ starting with some M_{-1}. Each approximation M_k with $k \geq 0$ contributes a transition relation $T_k(M_k)$ which can be combined into a limit M_ω. If Γ depends on the past w. r. t. the class of epistemic transition structures from which the approximands are constructed and which also contains the limit, then the interpretation of the limit M_ω yields a fixed point.

Proposition 2. *Let $\Gamma = (\mathsf{B}, \mathcal{T})$ be an epistemically guarded transition system over Σ, let $\mathcal{M} \subseteq \mathscr{M}_\Sigma(\mathsf{B})$ such that $\Gamma^M \in \mathcal{M}$ for every $M \in \mathcal{M}$ and $(\mathsf{B}, \bigcup_{0 \leq k} T_k(M_k)) \in \mathcal{M}$ for all $(M_k)_{0 \leq k} \subseteq \mathcal{M}$ with $T_k(M_{k'}) = T_k(M_k)$ for all $k' \geq k \geq 0$, and let Γ depend on the past w. r. t. $\mathcal{M}$. Let $M_{-1} \in \mathcal{M}$, $M_{i+1} = \Gamma^{M_i}$ for all $i \geq -1$, and $M_\omega = (\mathsf{B}, \bigcup_{0 \leq k} T_k(M_k))$. Then $\Gamma^{M_\omega} = \Gamma^{\Gamma^{M_\omega}}$.*

A sufficient criterion for obtaining a comprehensive class of epistemic transition structures $\mathcal{M}$ such that Γ depends on the past w. r. t. $\mathcal{M}$ is provided by epistemic witnesses: If some knowledge formula $\mathsf{K}_a\, \varphi$ of Γ does not hold at some state of an interpreting epistemic transition structure there is evidence in the past of this structure why it does not hold. Formally, a structure $M \in \mathscr{M}_{P,A}(\mathsf{B})$ *provides epistemic witnesses* for a formula $\mathsf{K}_a\, \varphi \in \Phi_{P,A}$ if for all $k \geq 0$, $s \in S_k(M)$ it holds that if $M, s \not\models \mathsf{K}_a\, \varphi$, then there is an $s' \in S_k(M)$ with $(s, s') \in E_a$ and $M, s' \not\models \varphi$.

Lemma 1. *Let $\Gamma = (\mathsf{B}, \mathcal{T})$ be an epistemically guarded transition system over Σ and let $\mathcal{M} \subseteq \mathscr{M}_\Sigma(\mathsf{B})$ such that all $M \in \mathcal{M}$ provide epistemic witnesses for all knowledge guards in Γ. Then Γ is depending on the past w. r. t. $\mathcal{M}$.*

A sufficient criterion, in turn, for a structure $M \in \mathscr{M}_{P,A}(S, E, L, S_0)$ to provide epistemic witnesses is M being *synchronous*: if for all $a \in A$ and all reachable $s_1 \in$

$S_{k_1}(M)$ and $s_2 \in S_{k_2}(M)$ with $(s_1, s_2) \in E_a$ it holds that $s_1, s_2 \in S_{\min\{k_1,k_2\}}(M)$. In a synchronous structure the temporal and the epistemic dimension for each agent are hence tightly coupled and agents cannot access the future, but also do not need to know the future.

Example 6. The interpretation M_{bt} of the bit transmission problem given in Ex. 2 provides epistemic witnesses, but is not synchronous: the sender S cannot distinguish z_0 reachable at depth 0 of M_{bt} from z_1 that is only reachable at depth 1, and similarly the receiver R cannot distinguish z_1 from z_3 at the respective depths of 1 and 2. □

An epistemically guarded transition system $\Gamma = (\mathsf{B}, \mathcal{T})$ over Σ *provides epistemic witnesses* if for each $M \in \mathscr{M}_\Sigma(\mathsf{B})$ the interpretation Γ^M provides epistemic witnesses for all knowledge formulæ occurring in some of the action guards of Γ; Γ is *synchronous* if each Γ^M is synchronous. Moreover, Γ can syntactically be seen to be synchronous (cf. [14, p. 135]) if it is round-based where all agents perform some action in each round and record locally which actions they have taken.

4 (Re-)Interpreting Knowledge-based Programs

The results by Fagin et al. [13,14] guarantee a unique interpretation for all synchronous knowledge-based programs; the approach by De Haan et al. [10] aims at extending the interpretation to asynchronous programs, but assigns semantics also to contradictory or self-fulfilling programs.

The necessity of avoiding contradictory or self-fulfilling behaviour already occurs in the design of synchronous programming languages [6]: Their underlying principle is "perfect synchrony", that any reaction of a program takes zero time and that thus whatever is output in reaction to some input is already present at the same time as the input. Since the presence or absence of signals can be tested, this requires "logical coherence" saying that a (non-input) signal is present in a reaction if, and only if, this signal is emitted in this very reaction. A program needs to be both *reactive* in the sense of leading to some logically coherent signal status, and *determinate*, i.e., not showing several such statuses. For example, in Esterel [7], the program fragment

```
present S then nothing else emit S end
```

is not reactive, but contradictory: signal S is only emitted if it is not emitted; and

```
present S then emit S else nothing end
```

is not determinate, but self-fulfilling: S is emitted if it is emitted, and it is not emitted if it is not. Such programs can be revealed by using a cycle-detecting static analysis, as is done in Lustre [18], or, for including more intricate cases, by Berry's "constructive semantics" as for Esterel [8]. Building on a "logical semantics" recording what is emitted in each step of execution, a *must*/*cannot* analysis is performed: what must/cannot be emitted, which branch must/cannot be executed. It is then required that for each signal it can be decided whether it must be present or it cannot be present. For example, in the parallel execution

```
   [ present S1 then emit S1 end ]
|| [ present S1 then present S2 then nothing else emit S2 end end ]
```

both signals can be emitted — if `S1` is assumed to be present, and `S2` absent —, but none must be emitted. Thus the constructive semantics does not reach a decision of what must/cannot be present and the program is not constructive. Intriguingly, however, there is exactly one coherent signal status that can be reached by execution: `S1` and `S2` absent.

We adapt Berry's constructive semantics approach to knowledge-based programs. In fact, the first, non-reactive Esterel program fragment resembles the variable setting problem described in Ex. 3(a), the second, non-determinate fragment directly corresponds to Ex. 4(d), and the last, combined fragment is essentially the same as Ex. 4(c). We first define a must/can version of epistemic transition structures with a lower (must) and an upper bound (can). Based on a positive (must) and negative (cannot) satisfaction relation of epistemic formulæ over these structures we show how an epistemically guarded transition system can be interpreted yielding another epistemic must/can transition structure. For uniformity, we rephrase this interpretation in terms of the negation normal form of formulæ and demonstrate that the constructive interpretation is always monotone and leads to a least fixed point. For any knowledge-based program, this fixed point soundly shows which executions are necessary and which are possible. However, the fixed point need not be decided, and more can be possible than is necessary. We show that synchronous programs always lead to decided fixed points.

4.1 Epistemic Must/Can Transition Structures

An *epistemic must/can transition structure* $Y = (S, E, L, S_0, (T_\mu, T_\nu))$ over $\Sigma = (P, A)$ is given by an epistemic state basis $\mathsf{B} = (S, E, L, S_0)$ and two *lower* and *upper transition relations* $T_\mu, T_\nu \subseteq S \times S$ with $T_\mu \subseteq T_\nu$. In particular, $Y_\mu = (\mathsf{B}, T_\mu)$ and $Y_\nu = (\mathsf{B}, T_\nu)$ are epistemic transition structures over Σ with $Y_\mu \subseteq Y_\nu$.

The *positive* and *negative satisfaction relations* of an epistemic formula $\varphi \in \Phi_{P,A}$ over the epistemic must/can transition structure Y at a state $s \in S_\omega(Y_\nu)$, written $Y, s \models_{\mathrm{p}} \varphi$ and $Y, s \models_{\mathrm{n}} \varphi$, are defined as follows:

$$Y, s \models_{\mathrm{p}} p \iff p \in L(s)$$
$$Y, s \not\models_{\mathrm{p}} \text{false}$$
$$Y, s \models_{\mathrm{p}} \neg\varphi \iff Y, s \models_{\mathrm{n}} \varphi$$
$$Y, s \models_{\mathrm{p}} \varphi_1 \wedge \varphi_2 \iff Y, s \models_{\mathrm{p}} \varphi_1 \text{ and } Y, s \models_{\mathrm{p}} \varphi_2$$
$$Y, s \models_{\mathrm{p}} \mathsf{K}_a\,\varphi \iff Y, s' \models_{\mathrm{p}} \varphi \text{ for all } s' \in S_\omega(Y_\nu) \text{ with } (s, s') \in E_a$$

$$Y, s \models_{\mathrm{n}} p \iff p \notin L(s)$$
$$Y, s \models_{\mathrm{n}} \text{false}$$
$$Y, s \models_{\mathrm{n}} \neg\varphi \iff Y, s \models_{\mathrm{p}} \varphi$$
$$Y, s \models_{\mathrm{n}} \varphi_1 \wedge \varphi_2 \iff Y, s \models_{\mathrm{n}} \varphi_1 \text{ or } Y, s \models_{\mathrm{n}} \varphi_2$$
$$Y, s \models_{\mathrm{n}} \mathsf{K}_a\,\varphi \iff Y, s' \models_{\mathrm{n}} \varphi \text{ for some } s' \in S_\omega(Y_\mu) \text{ with } (s, s') \in E_a$$

A formula is positively satisfied over Y if it must be true given the upper bound Y_ν of possible behaviour, it is negatively satisfied if it cannot be true given the lower bound Y_μ of necessary behaviour. In fact, it holds that what must be true can also be true:[4]

Lemma 2. *Let $Y = (S, E, L, S_0, (T_\mu, T_\nu))$ be an epistemic must/can transition structure over (P, A) and $\varphi \in \Phi_{P,A}$. Then for all $s \in S_\omega(Y_\nu)$, $Y, s \models_{\mathrm{p}} \varphi$ implies $Y, s \not\models_{\mathrm{n}} \varphi$.*

[4] The proofs are available in a long version at https://arxiv.org/abs/2301.10807.

The set of epistemic must/can transition structures over Σ and the epistemic state basis B is denoted by $\mathscr{Y}_{\Sigma}(\mathsf{B})$. We say that $Y_1 \sqsubseteq Y_2$ for $Y_1, Y_2 \in \mathscr{Y}_{\Sigma}(\mathsf{B})$ if $Y_{1,\mu} \subseteq Y_{2,\mu}$ and $Y_{1,\nu} \supseteq Y_{2,\nu}$: an *extension* raises the lower bound and reduces the upper bound.

As with epistemic transition structures, an epistemically guarded transition system $\Gamma = (S, E, L, S_0, \mathcal{T})$ over (P, A) can be interpreted over an epistemic must/can transition structure $Y \in \mathscr{Y}_{P,A}(S, E, L, S_0)$: The *interpretation* of a guarded action $(\varphi \supset B) \in \mathcal{T}$ w.r.t. to Y is given by the pair $(\varphi \supset B)^{Y} = ((\varphi \supset B)^{Y,\mu}, (\varphi \supset B)^{Y,\nu})$ with

$$(\varphi \supset B)^{Y,\mu} = \{(s, s') \in B \mid s \in S_{\omega}(Y_{\mu}) \text{ and } Y, s \models_{\mathrm{p}} \varphi\} ,$$
$$(\varphi \supset B)^{Y,\nu} = \{(s, s') \in B \mid s \in S_{\omega}(Y_{\nu}) \text{ and } Y, s \not\models_{\mathrm{n}} \varphi\} .$$

By Lem. 2 it holds that $\tau^{Y,\mu} \subseteq \tau^{Y,\nu}$ for each $\tau \in \mathcal{T}$. The *constructive interpretation* of Γ w.r.t. Y is given by the epistemic must/can transition structure

$$\Gamma^{Y} = (S, E, L, S_0, (\textstyle\bigcup_{\tau\in\mathcal{T}} \tau^{Y,\mu}, \bigcup_{\tau\in\mathcal{T}} \tau^{Y,\nu})) .$$

This is well defined, i.e., $(\Gamma^{Y})_{\mu} \subseteq (\Gamma^{Y})_{\nu}$. We call Y a *constructive solution* for Γ if $\Gamma^{Y} = Y$; a constructive solution is *decided* if $Y_{\mu} = Y_{\nu}$.

Again as with epistemic transition structures, this interpretation over epistemic must/can transition structures can be iterated for finally reaching a stable structure — and this time interpretation turns out to be monotone.

Example 7. (a) Re-consider the cycle-breaking variable setting problem of Ex. 3(b). We start the interpretation in $Y_{vsb,0} = (\mathsf{B}_{vs}, (\emptyset, S_{vs}^{2}))$ and successively obtain the following epistemic must/can transition structures:

τ	$\tau^{Y_{vsb,0}}$	$\tau^{Y_{vsb,1}}$	$\tau^{Y_{vsb,2}}$	$\tau^{Y_{vsb,3}}$
$\mathsf{K}_{\mathrm{a}} \neg(\mathrm{q}_1 \wedge \neg\mathrm{q}_2) \supset \{(\mathrm{s}_0, \mathrm{s}_1)\}$	$\emptyset$ $\{(\mathrm{s}_0, \mathrm{s}_1)\}$	$\emptyset$ $\emptyset$	$\emptyset$ $\emptyset$	$\emptyset$ $\emptyset$
$\mathrm{true} \supset \{(\mathrm{s}_0, \mathrm{s}_2)\}$	$\{(\mathrm{s}_0, \mathrm{s}_2)\}$ $\{(\mathrm{s}_0, \mathrm{s}_2)\}$	$\{(\mathrm{s}_0, \mathrm{s}_2)\}$ $\{(\mathrm{s}_0, \mathrm{s}_2)\}$	$\{(\mathrm{s}_0, \mathrm{s}_2)\}$ $\{(\mathrm{s}_0, \mathrm{s}_2)\}$	$\{(\mathrm{s}_0, \mathrm{s}_2)\}$ $\{(\mathrm{s}_0, \mathrm{s}_2)\}$
$\mathsf{K}_{\mathrm{a}} \neg(\neg\mathrm{q}_1 \wedge \mathrm{q}_2) \supset \{(\mathrm{s}_0, \mathrm{s}_3)\}$	$\emptyset$ $\{(\mathrm{s}_0, \mathrm{s}_3)\}$	$\emptyset$ $\{(\mathrm{s}_0, \mathrm{s}_3)\}$	$\{(\mathrm{s}_0, \mathrm{s}_3)\}$ $\{(\mathrm{s}_0, \mathrm{s}_3)\}$	$\{(\mathrm{s}_0, \mathrm{s}_3)\}$ $\{(\mathrm{s}_0, \mathrm{s}_3)\}$

Not only does it hold that ${\Gamma_{vsb}}^{Y_{vsb,3}} = Y_{vsb,3}$, but the interpretations indeed evolve monotonically w.r.t. $\sqsubseteq$. Moreover, the structure $Y_{vsb,3}$ is decided and everything what can happen also must happen, i.e., $(Y_{vsb,3})_{\mu} = (Y_{vsb,3})_{\nu}$.

(b) For the cyclic variable setting problem, see Ex. 1(b) and Ex. 3(a), the interpretation process is monotone, but only yields

τ	$\tau^{Y_{vs,0}}$	$\tau^{Y_{vs,1}}$
$\mathsf{K}_{\mathrm{a}} \neg(\mathrm{q}_1 \wedge \neg\mathrm{q}_2) \supset \{(\mathrm{s}_0, \mathrm{s}_1)\}$	$(\emptyset, \{(\mathrm{s}_0, \mathrm{s}_1)\})$	$(\emptyset, \{(\mathrm{s}_0, \mathrm{s}_1)\})$
$\mathsf{K}_{\mathrm{a}} \neg(\neg\mathrm{q}_1 \wedge \mathrm{q}_2) \supset \{(\mathrm{s}_0, \mathrm{s}_2)\}$	$(\emptyset, \{(\mathrm{s}_0, \mathrm{s}_2)\})$	$(\emptyset, \{(\mathrm{s}_0, \mathrm{s}_2)\})$

The epistemic must/can transition structure $Y_{vs,1}$ is not decided, and indeed there are two solutions of Γ_{vs} in terms of epistemic transition structures. However, the same undecidedness holds true for Γ_{nc} of Ex. 4(c), that is, the unique solution is also missed by the constructive interpretation. □

4.2 Constructive Interpretation

The separated positive (must) and negative (cannot) satisfaction relations over an epistemic must/can transition structure $Y \in \mathscr{Y}_{P,A}(S, E, L, S_0)$ can be merged into a single, uniform satisfaction relation relying on the *negation normal form* of epistemic formulæ where negation only occurs in front of propositions. For an arbitrary $\varphi \in \Phi_{P,A}$ there exists an equivalent $\mathrm{nnf}(\varphi) \in \Phi_{P,A}$ in negation normal form, such that, in particular

$$\begin{aligned} &\mathrm{nnf}(\neg p) = \neg p && \mathrm{nnf}(\neg\neg\varphi) = \mathrm{nnf}(\varphi) \\ &\mathrm{nnf}(\neg\mathrm{false}) = \mathrm{true} && \mathrm{nnf}(\neg(\varphi_1 \wedge \varphi_2)) = \mathrm{nnf}(\neg\varphi_1) \vee \mathrm{nnf}(\neg\varphi_2) \\ & && \mathrm{nnf}(\neg\mathsf{K}_a\,\varphi) = \mathsf{M}_a\,\mathrm{nnf}(\neg\varphi) \end{aligned}$$

The *constructive satisfaction relation* $Y, s \models \varphi$ for a state $s \in S_\omega(Y_\nu)$ and an epistemic formula $\varphi \in \Phi_{P,A}$ in negation normal form is defined just as for arbitrary epistemic formulæ, but using the upper bound Y_ν for the universal quantifier of K_a and the lower bound Y_μ for the existential quantifier of M_a; in particular,

$$\begin{aligned} &Y, s \models \neg p \iff p \notin L(s) \\ &Y, s \models \mathsf{K}_a\,\varphi \iff Y, s' \models \varphi \text{ f. a. } s' \in S_\omega(Y_\nu) \text{ with } (s, s') \in E_a \\ &Y, s \models \mathsf{M}_a\,\varphi \iff \text{ex. } s' \in S_\omega(Y_\mu) \text{ s. t. } (s, s') \in E_a \text{ and } Y, s' \models \varphi \end{aligned}$$

The constructive satisfaction relation indeed combines $\models_{\mathrm{p}}$ and $\models_{\mathrm{n}}$:

Lemma 3. *Let $Y \in \mathscr{Y}_{P,A}(\mathsf{B})$, $\varphi \in \Phi_{P,A}$, and $s \in S_\omega(Y_\nu)$. Then $Y, s \models_{\mathrm{p}} \varphi$ iff $Y, s \models \mathrm{nnf}(\varphi)$ and $Y, s \models_{\mathrm{n}} \varphi$ iff $Y, s \models \mathrm{nnf}(\neg\varphi)$.*

It follows that if $Y_\mu = Y_\nu$, then $Y, s \models \varphi$ if, and only if, $Y_\mu, s \models \varphi$ or, equivalently, $Y_\nu, s \models \varphi$. We also obtain that constructive satisfaction is preserved when extending epistemic must/can transition structures:

Lemma 4. *Let $Y, Y' \in \mathscr{Y}_{P,A}(\mathsf{B})$ with $Y \sqsubseteq Y'$ and let $\varphi \in \Phi_{P,A}$. Then $Y, s \models \mathrm{nnf}(\varphi)$ implies $Y', s \models \mathrm{nnf}(\varphi)$ for all $s \in S_\omega(Y'_\nu)$.*

This preservation of satisfaction yields that constructive interpretation is monotone.

Proposition 3. *Let $\Gamma = (\mathsf{B}, \mathcal{T})$ be an epistemically guarded transition system over Σ and $Y, Y' \in \mathscr{Y}_\Sigma(\mathsf{B})$ such that $Y \sqsubseteq Y'$. Then $\Gamma^Y \sqsubseteq \Gamma^{Y'}$.*

Finally, we can observe that $\mathscr{Y}_\Sigma(\mathsf{B})$ for $\mathsf{B} = (S, E, L, S_0)$ with the ordering $\sqsubseteq$ is an *inductive partial order*: each directed subset $\Delta \subseteq \mathscr{Y}_\Sigma(\mathsf{B})$ has a least upper bound $\bigsqcup \Delta$ w.r.t. $\sqsubseteq$, where *directed* means that every two $Y_1, Y_2 \in \Delta$ have an upper bound $Y \in \Delta$ such that $Y_1 \sqsubseteq Y$ and $Y_2 \sqsubseteq Y$; and there is also a *bottom* or least element $\bot_{\Sigma,\mathsf{B}} = (S, E, L, S_0, (\emptyset, S \times S)) \in \mathscr{Y}_\Sigma(\mathsf{B})$.

Proposition 4. $(\mathscr{Y}_\Sigma(\mathsf{B}), \sqsubseteq, \bot_{\Sigma,\mathsf{B}})$ *is an inductive partial order.*

Pataraia's fixed-point theorem [9, §8.22] now guarantees that the monotone operator $Y \mapsto \Gamma^Y$ for each epistemically guarded transition system $\Gamma = (\mathsf{B}, \mathcal{T})$ has a least fixed point in the inductive partial order. It can be computed by, possibly transfinite, iterated application of constructive interpretation to $\bot_{\Sigma,\mathsf{B}}$, that is, $Y_0 = \bot_{\Sigma,\mathsf{B}}$, $Y_{\alpha+1} = \Gamma^{Y_\alpha}$ for a successor ordinal $\alpha + 1$, and $Y_\lambda = \bigsqcup_{\alpha<\lambda} Y_\alpha$ until equality [9, Exc. 8.19]. Compared to the iteration semantics of Sect. 3.3, the computation of the constructive semantics thus does not have to record all previous approximations in order to find a repetition.

4.3 (Un-)Decided Constructive Fixed Points

If any constructive fixed point $Y = \Gamma^Y$ with $Y \in \mathscr{Y}_\Sigma(\mathsf{B})$ is decided, then there is the solution $Y_\mu = \Gamma^{Y_\mu} = \Gamma^{Y_\nu} = Y_\nu$ in terms of epistemic transition structures, and Γ is not contradictory. Even if it is not decided, the must/can structures $Y_{\mu\mu} = (\mathsf{B}, (T(Y_\mu), T(Y_\mu))) \in \mathscr{Y}_\Sigma(\mathsf{B})$ and $Y_{\nu\nu} = (\mathsf{B}, (T(Y_\nu), T(Y_\nu))) \in \mathscr{Y}_\Sigma(\mathsf{B})$ satisfy $Y \sqsubseteq Y_{\mu\mu}$ and $Y \sqsubseteq Y_{\nu\nu}$, such that by Prop. 3 we obtain $Y = \Gamma^Y \sqsubseteq \Gamma^{Y_{\mu\mu}}, \Gamma^{Y_{\nu\nu}}$ which yields $Y_\mu \subseteq \Gamma^{Y_\mu}$ and $\Gamma^{Y_\nu} \subseteq Y_\nu$, but not equality, in general. For the least constructive fixed point $\mu\Gamma$, any solution $M = \Gamma^M$ thus satisfies $(\mu\Gamma)_\mu \subseteq M \subseteq (\mu\Gamma)_\nu$, always giving sound lower and upper bounds and, if $\mu\Gamma$ is decided, moreover unique solvability:

Proposition 5. *Let $\Gamma = (\mathsf{B}, \mathcal{T})$ be an epistemically guarded transition system over Σ and assume $\mu\Gamma \in \mathscr{Y}_\Sigma(\mathsf{B})$ is decided. Then Γ has a unique solution in $\mathscr{M}_\Sigma(\mathsf{B})$.*

Still, even for epistemically guarded transition systems that provide epistemic witnesses it is not guaranteed that the least constructive fixed point is decided:

Example 8. Consider the following epistemically guarded transition system $\Gamma_{nd} = (\mathsf{B}_{nd}, \mathcal{T}_{nd})$ over $\Sigma_{nd} = (P_{nd}, A_{nd})$ with $P_{nd} = \{\mathrm{p}, \mathrm{q}\}$ and $A_{nd} = \{\mathrm{a}, \mathrm{b}\}$:

$$O_{nd,\mathrm{a}} = \{\mathrm{q}\} \qquad O_{nd,\mathrm{b}} = \emptyset$$

$$\begin{tikzcd} {} \arrow[r] & \overset{\mathrm{u}_0}{\boxed{\mathrm{p}, \neg\mathrm{q}}} \arrow[rr, "\mathsf{K}_\mathrm{b}\, \mathsf{M}_\mathrm{a}\, \mathrm{p}?"] & & \overset{\mathrm{u}_1}{\boxed{\mathrm{p}, \mathrm{q}}} \end{tikzcd}$$

Constructive interpretation yields the non-decided fixed point Y_{nd} with $T(Y_{nd,\mu}) = \emptyset$ and $T(Y_{nd,\nu}) = \{(\mathrm{u}_0, \mathrm{u}_1)\}$, as $Y_{nd}, \mathrm{u}_0 \not\models \mathsf{K}_\mathrm{b}\, \mathsf{M}_\mathrm{a}\, \mathrm{p}$, but also $Y_{nd}, \mathrm{u}_0 \not\models \mathsf{M}_\mathrm{b}\, \mathsf{K}_\mathrm{a} \neg\mathrm{p}$: the states u_0 and u_1 can be distinguished by agent a, and agent b cannot tell whether a step has been taken. In u_0 the formula $\mathsf{M}_\mathrm{a}\, \mathrm{p}$ holds w.r.t. Y_{nd}, but in u_1 it does not, since $(\mathrm{u}_1, \mathrm{u}_0) \notin E_{nd,a}$. On the other hand, Γ_{nd} provides epistemic witnesses pathologically, since ${\Gamma_{nd}}^M, s \models \mathsf{K}_\mathrm{b}\, \mathsf{M}_\mathrm{a}\, \mathrm{p}$ for any $M \in \mathscr{M}_{\Sigma_{nd}}(\mathsf{B}_{nd})$ and any $s \in S_\omega({\Gamma_{nd}}^M)$, and hence has a unique interpretation, which in this case is ${\Gamma_{nd}}^{Y_{nd,\mu}} = Y_{nd,\nu} = {\Gamma_{nd}}^{Y_{nd,\nu}}$. □

For synchronous epistemically guarded transition systems, however, the least fixed point is decided, since all knowledge refers to a past that must have happened:

Lemma 5. *Let $\Gamma = (\mathsf{B}, \mathcal{T})$ be an epistemically guarded transition system over Σ that is synchronous. Let $Y \in \mathscr{Y}_\Sigma(\mathsf{B})$ satisfy $\Gamma^Y = Y$. Then Y is decided.*

Summing up, the constructive approach to interpreting knowledge-based programs subsumes the solutions for synchronous programs and provides a sound procedure for obtaining lower and upper bounds for the execution of both synchronous and asynchronous programs. The approach, however, is not complete: If the least constructive fixed point $\mu\Gamma$ is undecided, a system Γ may be contradictory without any solution (see Ex. 3(a)), self-fulfilling with several solutions (see Ex. 4(d)), or it may have a unique solution in terms of epistemic transition structures (see Ex. 4(c)). One strategy that suggests itself for analysing Γ further is to check whether an interpretation using the lower bound $(\mu\Gamma)_\mu$ of the least fixed point satisfies $\Gamma^{(\mu\Gamma)_\mu} = (\mu\Gamma)_\nu = \Gamma^{(\mu\Gamma)_\nu}$, which means that when executing according to what must happen all what can happen is already covered (see Ex. 8).

5 Knowledge-based Programs as Rule Systems

The "executions" of an epistemically guarded transition system $\varGamma$ can be captured as derivations of two mutually dependent inductive rule systems, like used for inductive definitions [1,19]. One rule system defines the reachability in $\varGamma$, the other one the satisfaction of knowledge formulæ in negation normal form over $\varGamma$. When $\varGamma$ provides epistemic witnesses, the mutual dependence can be resolved by stratifying the rule system for reachability according to the depth of the execution. In the general case, the non-monotone dependence of the formula satisfaction system on the reachability system — the more states are reachable, the less is known — can be mitigated by extending the notion of rule systems to include also negative premisses: The conclusion of a rule is derivable if all its (positive) premisses are derivable, but none of its negative premisses. When applied to knowledge formulæ, negative premisses express that no counterexample is reachable.

The general rule systems can also be read as logic programs with "negation as failure" [11]. A direct application of the must/can approximation technique to the general rule system or, equivalently, the logic program resulting from a knowledge-based program reconstructs the Kripke-Kleene fixed point; the possible solutions correspond to "stable models" [16].

5.1 Inductive Rule Systems

An *inductive rule system* R consists of *rules* of the form X/y where the *premisses* $X \subseteq U$ and the *conclusion* $y \in U$ are drawn from some *universe* of *judgements* U. A rule X/y is interpreted as "if all X can be inferred, then y can be inferred". The *derivations* in R together with their *sets of premisses* and *conclusions* are inductively defined as follows:

– a $y \in U$ is itself a derivation; its set of premisses is $\{y\}$, its conclusion is y;
– if $X/y \in R$ and $(d_x)_{x \in X}$ a family of derivations with conclusions $(x)_{x \in X}$, then $(d_x)_{x \in X}/y$ is a derivation; its set of premisses is the union of the premisses of $(d_x)_{x \in X}$, its conclusion is y.

A $y \in U$ is *derivable* in R if there is a derivation in R with the empty set of premisses and conclusion y. The set of derivable conclusions of R coincides with the least fixed point $\mu\hat{R}$ of $\hat{R}$: $\wp U \to \wp U$ defined by $\hat{R}(P) = \{y \in U \mid \text{ex. } X/y \in R \text{ s.t. } X \subseteq P\}$.

In logic programming terms, a rule $X/y \in R$ yields a Horn clause $y \leftarrow X$ [11]. The least fixed point $\mu\hat{R}$ coincides with minimal Herbrand model of the logic program corresponding to R and thus with the single stable model, as no negation is involved [11,16].

For expressing reachability and the satisfaction of knowledge formulæ in an epistemically guarded transition system $\varGamma = (S, E, L, S_0, \mathcal{T})$ over (P, A) as inductive rule systems, we use two types of judgements, one of the form $s \in^{\varGamma} S_\omega$ with $s \in S$ for "state s is reachable in $\varGamma$", and one of the form $s \models^{\varGamma} \varphi$ with $s \in S$ and $\varphi \in \varPhi_{P,A}$ in negation normal form for "state s satisfies formula φ in $\varGamma$". The rules for reachability read:

$$\frac{}{s_0 \in^{\varGamma} S_\omega} \quad \text{if } s_0 \in S_0 \qquad \frac{s \in^{\varGamma} S_\omega}{s' \in^{\varGamma} S_\omega} \quad \begin{array}{l}\text{if ex. } (\varphi \supset B) \in \mathcal{T},\\ (s, s') \in B, \text{ and } s \models^{\varGamma} \varphi\end{array}$$

where $s \models^\Gamma \varphi$ in the side condition of the second rule requires this judgement to be derivable in the rule system for satisfaction. The rules for this system read:

$$\frac{}{s \models^\Gamma \mathrm{true}}\ \text{if } s \in^\Gamma S_\omega \qquad \frac{}{s \models^\Gamma p}\ \begin{array}{c}\text{if } s \in^\Gamma S_\omega,\\ p \in L(s)\end{array} \qquad \frac{}{s \models^\Gamma \neg p}\ \begin{array}{c}\text{if } s \in^\Gamma S_\omega,\\ p \notin L(s)\end{array}$$

$$\frac{s \models^\Gamma \varphi_1 \quad s \models^\Gamma \varphi_2}{s \models^\Gamma \varphi_1 \wedge \varphi_2} \qquad \frac{s \models^\Gamma \varphi_1}{s \models^\Gamma \varphi_1 \vee \varphi_2} \qquad \frac{s \models^\Gamma \varphi_2}{s \models^\Gamma \varphi_1 \vee \varphi_2}$$

$$\frac{s' \models^\Gamma \varphi}{s \models^\Gamma \mathsf{M}_a\, \varphi}\ \begin{array}{c}\text{if } (s, s') \in E_a,\\ s' \in^\Gamma S_\omega\end{array} \qquad \frac{(s' \models_\Gamma \varphi)_{s' \in^\Gamma S_\omega,\ (s,s') \in E_a}}{s \models^\Gamma \mathsf{K}_a\, \varphi}$$

Here, the last rule for satisfaction in fact is not monotone w.r.t. reachability: In order to infer $s \models^\Gamma \mathsf{K}_a\, \varphi$ it is not necessary to infer $s' \models^\Gamma \varphi$ for all s' with $(s, s') \in E_a$, but only for those for which $s' \in^\Gamma S_\omega$ can be deduced — and also for all of those.

The notion of providing epistemic witnesses allows to stratify the inductive rule systems according to the involved depth $k \geq 0$: We specialise the judgement $s \in^\Gamma S_\omega$ into $s \in^\Gamma S_k$ meaning "state s is reachable in Γ in up to k steps" and, similarly, the judgement $s \models^\Gamma \varphi$ into $s \models^\Gamma_k \varphi$ meaning "formula φ is satisfied in Γ at state s considering states reachable in up to k steps". The rules for reachability become for all $k \geq 0$:

$$\frac{}{s_0 \in^\Gamma S_k}\ \text{if } s_0 \in^\Gamma S_0 \qquad \frac{s \in^\Gamma S_k}{s' \in^\Gamma S_{k+1}}\ \begin{array}{c}\text{if ex. } (\varphi \supset B) \in \mathcal{T},\\ (s, s') \in B, \text{ and } s \models^\Gamma_k \varphi\end{array}$$

Analogously the rules for satisfaction become for all $k \geq 0$:

$$\frac{}{s \models^\Gamma_k \mathrm{true}}\ \text{if } s \in^\Gamma S_k \qquad \frac{}{s \models^\Gamma_k p}\ \begin{array}{c}\text{if } s \in^\Gamma S_k,\\ p \in L(s)\end{array} \qquad \frac{}{s \models^\Gamma_k \neg p}\ \begin{array}{c}\text{if } s \in^\Gamma S_k,\\ p \notin L(s)\end{array}$$

$$\frac{s \models^\Gamma_k \varphi_1 \quad s \models^\Gamma_k \varphi_2}{s \models^\Gamma_k \varphi_1 \wedge \varphi_2} \qquad \frac{s \models^\Gamma_k \varphi_1}{s \models^\Gamma_k \varphi_1 \vee \varphi_2} \qquad \frac{s \models^\Gamma_k \varphi_2}{s \models^\Gamma_k \varphi_1 \vee \varphi_2}$$

$$\frac{s' \models^\Gamma_k \varphi}{s \models^\Gamma_k \mathsf{M}_a\, \varphi}\ \begin{array}{c}\text{if } (s, s') \in E_a,\\ s' \in^\Gamma S_k\end{array} \qquad \frac{(s' \models^\Gamma_k \varphi)_{s' \in^\Gamma S_k,\ (s,s') \in E_a}}{s \models^\Gamma_k \mathsf{K}_a\, \varphi}$$

In particular, the rules for $s \models^\Gamma_k \mathsf{M}_a\, \varphi$ and $s \models^\Gamma_k \mathsf{K}_a\, \varphi$ are sound for epistemically guarded transition systems providing epistemic witnesses. The notion of "providing epistemic witnesses" requires that, if $\mathsf{K}_a\, \varphi$ does not hold at depth k, there is a counterexample to φ at depth $\leq k$. The general case can be covered by dropping the depths and taking into account that $\mathsf{K}_a\, \varphi$ does not hold at some state s if, and only if, there is some reachable, a-indistinguishable state s' at which φ does not hold. Therefore, in order to derive that $\mathsf{K}_a\, \varphi$ indeed holds at some reachable state s, it is necessary and sufficient to show that it is *not* possible to derive that $\neg\varphi$ holds at some reachable, a-indistinguishable state s'.

5.2 General Rule Systems with Positive and Negative Premisses

For expressing negative information in terms of a rule system, we complement the positive premisses of the rules by negative ones: We consider general *rule systems* R over a universe U consisting of rules of the form $(X, \nVdash Z)/y$ where $X, Z \subseteq U$ are the *positive* and *negative premisses*, and $y \in U$ is the *conclusion*; it is interpreted as "if all X can be inferred but no Z, then y can be inferred". The *derivations* in R together with their *sets of positive and negative premisses* and *conclusions* are again inductively defined as follows:

– a $y \in U$ is itself a derivation; its set of positive premisses is $\{y\}$, its set of negative premisses is $\emptyset$, and its conclusion is y;
– if $(X, \nVdash Z)/y \in R$ and $(d_x)_{x \in X}$ a family of derivations with conclusions $(x)_{x \in X}$, then $((d_x)_{x \in X}, \nVdash Z)/y$ is a derivation; its set of positive premisses is the union of the positive premisses of $(d_x)_{x \in X}$, its set of negative premisses is the union of the negative premisses of $(d_x)_{x \in X}$ together with Z, and its conclusion is y.

For a $B \subseteq U$, let $\bar{R}(B)$ be all those $y \in U$ such that there is a derivation of y in R with the empty set of positive premisses and no negative premisses in B. The set of *derivable conclusions* of R is given by the least fixed point of $\bar{R}$ if it exists.

From the logic programming perspective, a general rule $(X, \nVdash Z)/y \in R$ can be seen as a clause of the form $y \leftarrow X, \nVdash Z$ with $\nVdash$ read as "negation as failure" [5,11]. Checking that a $B \subseteq U$ is a "stable model" of the logic program obtained from R in this way corresponds to the following process on general rule systems: first the reduct R_B is formed by disregarding all rules $(X, \nVdash Z)/y \in R$ with $B \cap Z \neq \emptyset$ and transforming the remaining rules $(X, \nVdash Z)/y \in R$ into $X/y \in R_B$; then R_B is an inductive rule system and B is stable if $B = \mu\hat{R}_B$. In particular, the stable models correspond to the *solutions* of $\bar{R}(B) = B$.

With this generalised notion of rule systems we can reformulate and combine the two inference systems for reachability and satisfaction in an epistemically guarded transition system $\Gamma = (S, E, L, S_0, \mathcal{T})$ over (P, A) by using a single judgement $s \models^{\Gamma}_{\omega} \varphi$ for "state s satisfies φ in Γ and state s is reachable in Γ". A negative premiss $\nVdash(s \models^{\Gamma}_{\omega} \text{true})$ thus stands for "$s \in^{\Gamma} S_{\omega}$ cannot be deduced". The new rules with also negative premisses read:

$$\frac{}{s_0 \models^{\Gamma}_{\omega} \text{true}} \ \text{if } s_0 \in S_0 \qquad \frac{s \models^{\Gamma}_{\omega} \varphi}{s' \models^{\Gamma}_{\omega} \text{true}} \ \begin{array}{l}\text{if ex. } (\varphi \supset B) \in \mathcal{T},\\ \quad (s, s') \in B\end{array}$$

$$\frac{s \models^{\Gamma}_{\omega} \text{true}}{s \models^{\Gamma}_{\omega} p} \ \text{if } p \in L(s) \qquad \frac{s \models^{\Gamma}_{\omega} \text{true}}{s \models^{\Gamma}_{\omega} \neg p} \ \text{if } p \notin L(s)$$

$$\frac{s \models^{\Gamma}_{\omega} \varphi_1 \quad s \models^{\Gamma}_{\omega} \varphi_2}{s \models^{\Gamma}_{\omega} \varphi_1 \wedge \varphi_2} \qquad \frac{s \models^{\Gamma}_{\omega} \varphi_1}{s \models^{\Gamma}_{\omega} \varphi_1 \vee \varphi_2} \qquad \frac{s \models^{\Gamma}_{\omega} \varphi_2}{s \models^{\Gamma}_{\omega} \varphi_1 \vee \varphi_2}$$

$$\frac{s' \models^{\Gamma}_{\omega} \varphi}{s \models^{\Gamma}_{\omega} \mathsf{M}_a\, \varphi} \ \text{if } (s, s') \in E_a \qquad \frac{s \models^{\Gamma}_{\omega} \text{true} \quad \nVdash(s' \models^{\Gamma}_{\omega} \text{nnf}(\neg\varphi))_{(s,s') \in E_a}}{s \models^{\Gamma}_{\omega} \mathsf{K}_a\, \varphi}$$

The rule for $s \models^{\Gamma}_{\omega} \mathsf{K}_a\, \varphi$ checks that s is reachable, but that no counterexample to φ can be reached at an a-undistinguishable state.

Using general rule systems, the solvability of an epistemically guarded transition system is shifted to computing derivable conclusions. As for knowledge-based programs, it is not obvious from just the rules of a system R whether there are solutions of $\bar{R}(B) = B$ at all, and whether there is a least one.

Example 9. (a) The general rule system

$$R_0 = \left\{ \frac{x_1}{x_1}, \frac{\nVdash x_1 \ \nVdash x_2}{x_2} \right\} \quad \text{over} \quad \{x_1, x_2\}$$

has no set of derivable conclusions, since $\bar{R}_0$ has no fixed point; in particular, $\bar{R}_0(\emptyset) = \{x_2\}$ and $\bar{R}_0(\{x_1\}) = \emptyset = \bar{R}_0(\{x_2\})$. In terms of stable models, computing $\bar{R}_0(\emptyset)$ amounts to removing the negative premisses from the rule $(\emptyset, \nVdash\{x_1, x_2\})/x_2$, such that the inductive rules $\{x_1\}/x_1$ and $\emptyset/x_2$ remain; and computing $\bar{R}_0(\{x_i\})$ leads to the single inductive rule $\{x_1\}/x_1$ for $i \in \{1, 2\}$.

R_0 also demonstrates that the set of derivable conclusions of a general rule system R need not coincide with the least fixed point of the operator $\hat{R}\colon \wp U \to \wp U$ when transferred from inductive rule systems by now setting $\hat{R}(P) = \{y \in U \mid \text{ex. } (X, \nVdash Z)/y \in R \text{ s.t. } X \subseteq P, P \cap Z = \emptyset\}$: $\mu\hat{R}_0 = \{x_1\}$.

On the other hand, in view of the general rule system for epistemically guarded transition systems R_0 can also be rephrased as a knowledge-based program with a single agent a and a single variable $\mathrm{x} \in \{0, 1, 2\}$, which a cannot observe, started with $\mathrm{x} = 0$:

```
if $\mathsf{M}_\mathrm{a}\, \mathrm{x} = 1 \rightarrowtriangle \mathrm{x} \leftarrow 1$
  $[\!]$ $\mathsf{K}_\mathrm{a}(\mathrm{x} \neq 1 \wedge \mathrm{x} \neq 2) \rightarrowtriangle \mathrm{x} \leftarrow 2$ fi
```

(b) There may be several solutions of a general rule system, but no least one:

$$R_1 = \left\{ \frac{\nVdash x_1}{x_3}, \frac{\nVdash x_3}{x_1} \right\} \quad \text{over} \quad \{x_1, x_3\}$$

has the solutions $\{x_1\}$ and $\{x_3\}$, but $\emptyset$ is no solution. It corresponds to the "variable setting" knowledge-based program of the introduction, see Ex. 1(b):

```
if $\mathsf{K}_\mathrm{a}\, \mathrm{x} \neq 1 \rightarrowtriangle \mathrm{x} \leftarrow 3$
  $[\!]$ $\mathsf{K}_\mathrm{a}\, \mathrm{x} \neq 3 \rightarrowtriangle \mathrm{x} \leftarrow 1$ fi
```

(c) Combining a contradictory rule $(\emptyset, \nVdash\{x_1, x_2\})/x_2$ with the non-determined rules of R_1 we obtain the rule system

$$R_2 = \left\{ \frac{\nVdash x_1}{x_3}, \frac{\nVdash x_3}{x_1}, \frac{\nVdash x_1 \ \nVdash x_2}{x_2} \right\} \quad \text{over} \quad \{x_1, x_2, x_3\}$$

which has the unique solution $\{x_1\}$: if x_3 were inferable, i.e., x_1 not inferable, this would trigger the contradictory rule for x_2 (see Ex. 4(c)). □

5.3 Solving General Rule Systems

The observations and definitions for epistemic must/can transition structures and constructive interpretation, see Sect. 4.2, can now readily be transferred to a more abstract account for general rule systems. In fact, this reconstructs the "Kripke-Kleene fixpoint" using under- and over-approximations [11], though now using an inductive partial order. We also relate the case where the constructive interpretation is not only monotone, but continuous to knowledge-based programs.

Define, for a universe U, the set $\wp^{\pm}U$ as $\{(P, Q) \in \wp U \times \wp U \mid P \subseteq Q\}$ and the relation $\subseteq^{\pm} \subseteq \wp^{\pm}U \times \wp^{\pm}U$ as $(P, Q) \subseteq^{\pm} (P', Q')$ if, and only if, $P \subseteq P'$ and $Q \supseteq Q'$.

Lemma 6. *$(\wp^{\pm}U, \subseteq^{\pm}, \bot^{\pm}_{U})$ with $\bot^{\pm}_{U} = (\emptyset, U)$ is an inductive partial order.*

For a general rule system R over U with positive and negative premisses define the operator $\check{R}\colon \wp^{\pm}U \to \wp^{\pm}U$ that describes what *must* and what *can* be derived given what is assumed to be definitely and potentially derivable:

$$\begin{aligned}\check{R}(P, Q) = (&\{y \in U \mid \text{ex. } (X, \nVdash Z)/y \in R \text{ s.t. } X \subseteq P,\ Q \cap Z = \emptyset\},\\ &\{y \in U \mid \text{ex. } (X, \nVdash Z)/y \in R \text{ s.t. } X \subseteq Q,\ P \cap Z = \emptyset\})\end{aligned}$$

This is well-defined: if $(P, Q) \in \wp^{\pm}U$, then $\check{R}(P, Q) \in \wp^{\pm}U$, since for $P \subseteq Q$ and each $(X, \nVdash Z)/y \in R$ with $X \subseteq P$ and $Q \cap Z = \emptyset$ it holds that $X \subseteq Q$ and $P \cap Z = \emptyset$. The operator is always monotone:

Lemma 7. *Let R be a rule system over U. If $(P_1, Q_1) \subseteq^{\pm} (P_2, Q_2)$, then $\check{R}(P_1, Q_1) \subseteq^{\pm} \check{R}(P_2, Q_2)$.*

As for constructive interpretation, Pataraia's fixed-point theorem now guarantees that the monotone operator $\check{R}$ on the inductive partial order $(\wp^{\pm}U, \subseteq^{\pm}, \bot^{\pm}_{U})$ has a least fixed point. Again, it can be "computed" by possibly transfinite iterated application of $\check{R}$ to $\bot^{\pm}_{U}$. If, however, $\check{R}$ is even continuous, then, by Kleene's fixed-point theorem, it suffices to consider all finite approximations, i.e., $\mu\check{R} = \bigcup^{\pm}_{n\in\mathbb{N}} \check{R}^{n}(\bot^{\pm}_{U})$; that $\check{R}$ is *continuous* means that if $\Delta \subseteq \wp^{\pm}U$ is directed, then $\bigcup^{\pm} \check{R}(\Delta) = \check{R}(\bigcup^{\pm} \Delta)$.

Lemma 8. *Let R be a rule system over U such that every rule of R has only finitely many positive and negative premisses. Then $\check{R}$ is continuous.*

The rule system for an epistemically guarded transition system $\Gamma = (S, E, L, S_0, \mathcal{T})$ over (P, A) always has only finitely many positive premisses; if for each $s \in S$ and each $a \in A$ the set $\{s' \in S \mid (s, s') \in E_a\}$ is finite, then there are also only finitely many negative premisses, such that the corresponding must/can operator is continuous.

6 Reasoning About Knowledge-based Programs

We have implemented the constructive interpretation of knowledge-based programs in the prototypical "Temporal Epistemic Model Interpreter and Checker" (tEmIc[5]). The tool first computes the least constructive fixed point of a (finite state) epistemically guarded transition system. If the least fixed point is decided, the least solution in terms of epistemic transition structures has been found; otherwise it is checked whether the re-interpretation using the lower bound of the undecided least fixed point yields a solution.

[5] https://bitbucket.org/knappale/temic

If either succeeds, properties of the resulting model can be checked. These properties can be expressed in CTLK, the combination of the branching "Computation Tree Logic" (CTL) and epistemic logic [21]. What is more, CTLK can also be used in tEmIc for the action guards. The constructive interpretation just evaluates each universal quantifier of a CTL formula — A for "on all paths" — over the upper bound and each existential quantifier — E for "on some path" — over the lower bound. This adds the temporal dimension to the domain of application of knowledge-based programs. For the run-based interpreted systems of Fagin et al. [13], Van der Hoek and Woolridge [20] and Su [27] provide transformations for linear-time model checking based on local propositions, though for a fixed set of runs that does not depend on the evaluation of knowledge guards. The CTLK-model checker MCMAS [21] similarly operates on a fixed, predetermined model. In dynamic epistemic logic and its model checker DEMO [31], the transition structure is given by epistemic actions.

We first recapitulate briefly CTLK and then show its constructive evaluation over epistemic must/can transition structures. We next describe tEmIc by means of the bit transmission problem and the small paradoxical exercise of the "unexpected examination"; the tEmIc distribution also contains specifications for the well-known problems "Muddy Children" [31, pp. 93ff.] and "Sum-and-Product" [31, pp. 96f.]. Finally, we proceed to an application where CTLK is also used in the action guards: the Java memory model.

6.1 CTLK

The *CTLK-formulæ* over (P, A) are defined by the following grammar:

$$\varphi ::= p \mid \text{false} \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{K}_a\,\varphi \mid \mathsf{EX}\,\varphi \mid \mathsf{EG}\,\varphi \mid \mathsf{E}[\varphi_1\,\mathsf{U}\,\varphi_2]$$

where $p \in P$ and $a \in A$. The path quantifier E is interpreted as "there is a path", the temporal modality X as "in the next step", G as "always", and U as "until". We also consider the path quantifier A for "on all paths" and the modalities F for "eventually" and R for "release", such that $\neg\mathsf{EG}\,\neg\varphi$ is abbreviated by $\mathsf{AF}\,\varphi$ and $\neg\mathsf{E}[\neg\varphi_1\,\mathsf{U}\,\neg\varphi_2]$ by $\mathsf{A}[\varphi_1\,\mathsf{R}\,\varphi_2]$. The *satisfaction relation* $M, s \models \varphi$ of a CTLK-formula φ over (P, A) at state $s \in S$ of an epistemic transition structure $M = (S, E, L, S_0, T)$ over (P, A) conservatively extends the satisfaction relation of epistemic formulæ by

$$\begin{aligned}
M, s \models \mathsf{EX}\,\varphi &\iff \text{ex. } s_0, s_1, \ldots \in \mathscr{P}(M, s) \text{ s.t. } M, s_1 \models \varphi \\
M, s \models \mathsf{EG}\,\varphi &\iff \text{ex. } s_0, s_1, \ldots \in \mathscr{P}(M, s) \text{ s.t. } M, s_i \models \varphi \text{ f.a. } i \in \mathbb{N} \\
M, s \models \mathsf{E}[\varphi_1\,\mathsf{U}\,\varphi_2] &\iff \text{ex. } s_0, s_1, \ldots \in \mathscr{P}(M, s) \text{ and } l \in \mathbb{N} \text{ s.t.} \\
&\qquad M, s_i \models \varphi_1 \text{ f.a. } 0 \le i < l \text{ and } M, s_l \models \varphi_2
\end{aligned}$$

where $\mathscr{P}(M, s)$ denotes all *paths* of M, i.e., the infinite state sequences $s_0, s_1, \ldots \in S$ with $s_0 = s$ and $(s_i, s_{i+1}) \in T$ for all $i \in \mathbb{N}$. A CTLK-formula φ is *valid* in M, written $M \models \varphi$, if it is satisfied in all initial states, i.e., $M, s_0 \models \varphi$ for all $s_0 \in S_0(M)$.

For a direct definition of the satisfaction of CTLK-formulæ with an A, the existential path quantification for E has to be replaced by universal path quantification. As for simple epistemic logic, CTLK including $\mathsf{AX}\,\varphi$, $\mathsf{AG}\,\varphi$ etc. admits a negation normal form (see, e.g., [3, pp. 333f.]). The *constructive satisfaction relation* of a CTLK-formula in negation normal form over an epistemic must/can transition structure $Y = (S, E, L, S_0, \mathcal{T})$ over (P, A) at a state $s \in S_\omega(Y_\nu)$, written $Y, s \models \varphi$, conservatively extends the constructive satisfaction relation of epistemic formulæ and interprets E over the lower bound Y_μ and A over the upper bound Y_ν such that, in particular,

$$Y, s \models \mathsf{EF}\,\varphi \iff \text{ex. } s_0, s_1, \ldots \in \mathscr{P}(Y_\mu, s) \text{ and } i \in \mathbb{N} \text{ s. t. } Y, s_i \models \varphi$$
$$Y, s \models \mathsf{AF}\,\varphi \iff \text{f. a. } s_0, s_1, \ldots \in \mathscr{P}(Y_\nu, s) \text{ ex. } i \in \mathbb{N} \text{ s. t. } Y, s_i \models \varphi$$

6.2 tEmIc

tEmIc is a symbolic model interpreter and checker for epistemically guarded transition systems using CTLK. It is written in Java and uses binary decision diagrams for state space representation [28]; it also supports bounded integers and their arithmetic. Given a specification, tEmIc first computes the least constructive fixed point by iterated must/can interpretation. If this fixed point is not decided it checks whether another interpretation using the lower bound of the fixed point yields a solution. If either succeeds, tEmIc proceeds with model checking given properties; these statements can be specified as CTLK-formulæ which have to hold in all initial states or as a reachability query. Reachable deadlock states without outgoing transitions result in a warning.

For example, the bit transmission problem of the introduction as formalised in Ex. 1(a) can be represented as a tEmIc specification as follows (rules are introduced by keyword `action` followed by a name of the rule and the rule definition):

```
var sbit, ack, rbit, snt : boolean initial (ack | rbit | snt) <-> false;

agent S = { sbit, ack };  agent R = { rbit, snt };
let R_knows_bit = exists bit:boolean . K[R] sbit <-> bit;

action S_sends_bit_ok
guard not K[S] R_knows_bit do rbit := sbit, snt := true;
action S_sends_bit_failed
guard not K[S] R_knows_bit do ;
action R_sends_ack_ok
guard R_knows_bit and not K[R] K[S] R_knows_bit do ack := true;
action R_sends_ack_failed
guard R_knows_bit and not K[R] K[S] R_knows_bit do ;
```

Constructive interpretation yields in a few milliseconds the decided least fixed point of Ex. 2, over which some CTLK-properties can be checked:

```
check initial EF            R_knows_bit;
check initial EF       K[S] R_knows_bit;
check initial EF K[R] K[S] R_knows_bit;
```

The first two are reported to hold, but the last does not since agent R cannot gather enough information to be sure that the bit has been received by agent S.

For another example, consider the "unexpected examination" paradox [10, Sect. 4.7, there called "unexpected hanging"] (for a detailed account see, e. g., [26, Sects. 5.2f.]): A class is told that within the next week there will be an exam, but it will be a surprise. The class might reason that the exam cannot happen on Friday, because if there has been no exam up to Thursday it will not be a surprise on Friday any more; by backward induction it might reason that there cannot be a surprise exam in the next week at all. This problem statement can be readily expressed as a tEmIc specification:

```
var day : 0..5 initial day = 0;
var exam : 0..4;
var written : boolean initial written <-> false;

agent P = { day, written };

action act1
guard day < 5 and (day = exam) and (not K[P] day = exam) and not written
do written := true, day := day+1;
action act2
guard day < 5 and (day != exam) do day := day+1;
action stutter
do ;
```

Again, constructive interpretation yields in a few milliseconds a decided least fixed point. Over this epistemic transition structure we can check that on, e. g., Wednesday the exam can be written and still is indeed a surprise:

```
check reachable exam = 2 & written;
```

For such a reachability check TEMIC also provides a witness that tells that `act2` is executed twice after which `act1` follows. The following CTLK-property, however, is not satisfied, as it would have to hold in all initial states — and with `exam` being 4 the class cannot be surprised any more:

```
check initial EF written;
```

6.3 Memory Models

Memory models regulate the interaction between threads, their caches, and the main memory [23]. The original Java memory model — one of the first formal such models — has been harshly criticised for making several compiler optimisations impossible and has subsequently been superseded by a more liberal model [17, Ch. 17]. Keeping strong guarantees for sequentially consistent, well-synchronised programs, reorderings of data-independent statements or early, "prescient" reads from other threads are allowed for programs with data races. Still, some limits, like consistency with data or control flow dependencies or no "out-of-thin-air" values, should be in force [25,2].

For example, in the following two-threaded Java-like program to the left it should be possible that both thread-local registers `r1` and `r2` are assigned the value 1 when reading the global, shared variables `x` and `y`: A compiler could reorder the data-independent statements in both threads. This behaviour, however, should be forbidden in the example to the right, since there is a symmetric data dependence.

```
    x = y = 0                    x = y = 0
   r1 = r2 = 0                  r1 = r2 = 0
-----------------        ---------------------------
r1 = x; || r2 = y;       r1 = x;        || r2 = y;
y = 1;  || x = 1;        if (r1 == 1)   || if (r2 == 1)
-----------------          y = 1;       ||   x = 1;
  r1 = r2 = 1?           ---------------------------
                                r1 = r2 = 1?
```

We want to capture the behaviour of a multi-threaded (Java) program with a liberal memory model without having to check all possible compiler transformations — the correctness of such transformations would actually depend on the program semantics including the memory model. In fact, in the current Java memory model out-of-order executions have to be justified by other legal executions. We interpret these justifications as witnesses in terms of knowledge-based programs; our current exposition, however, neglects synchronisation. We first represent the state space of a two-threaded (Java) program like the ones above by the following tEmIc declarations:

```
var x, y, r1, r2 : 0..2 initial x = 0 & y = 0 & r1 = 0 & r2 = 0;
var step1, step2 : 1..3 initial step1 = 1 & step2 = 1;

agent t1 = { step1, r1 };  agent t2 = { step2, r2 };
```

The thread agents `t1` and `t2` can only observe their local registers and their program counters. The program steps for both threads are turned into actions like

```
action t1_1 guard step1 = 1 do r1 := x, step1 := step1+1;
action t1_2 guard step1 = 2 do y := 1, step1 := step1+1;
```

Additionally, we allow for a "prescient reading" of the value v from the main memory variable x by thread θ into the local variable r at step s by the following action:

```
action readθ_x_v_r_s
guard stepθ = s and K[θ] (EF (r = 0 & x = v) and EF (r = v & x = v))
do r := v, stepθ := stepθ+1;
```

The thread θ can read v from x into r early on if it *knows* that *there is an execution* where x has value v without dependence on already setting r to v, and, furthermore, that *there is an execution* where the early setting is confirmed. The statement `r1 = x;` of the first thread is expanded into three read actions `read1_x_0_r1_1`, `read1_x_1_r1_1`, and `read1_x_2_r1_1` plus the plain reading action `t1_1`. With this encoding, tEmIc reports that for the first example to the left it is indeed possible to obtain $\texttt{r1} = \texttt{r2} = 1$ in the least constructive fixed point, but that this is impossible for the example to the right.

A more intriguing case is presented by the following two examples: According to Manson et al. [23, pp. 35f.] (cf. also [2]), the program to the left can result in $\texttt{r1} = \texttt{r2} = \texttt{r3} = 1$:

```
        x = y = 0                              x = y = 0
     r1 = r2 = r3 = 0                      r1 = r2 = r3 = 0
 ---------------------------     -----------------------------------------
 r1 = x;          || r3 = y;     r1 = x;          || r2 = x; || r3 = y;
 if (r1 == 0)     || x = r3;     if (r1 == 0)     || y = r2; || x = r3;
   x = 1;         ||               x = 1;         ||         ||
 r2 = x;          ||             -----------------------------------------
 y = r2;          ||                       r1 = r2 = r3 = 1?
 ---------------------------
     r1 = r2 = r3 = 1?
```

A compiler could see that only 0 and 1 are possible for `x` and `y` and "can then replace `r2 = x` by `r2 = 1`, because either 1 was read from `x` on line 1 and there is no intervening write, or 0 was read from `x` on line 1, 1 was assigned to `x` on line 3, and there was no intervening write"; this definite assignment can be used to transform the last line to `y = 1;` which finally can be made the first action of the first thread, as there are no dependencies. But the same transformation is not possible for the program to the right, and there the same behaviour should be disallowed. Still, the left program is the result of inlining the second thread into the first. Our encoding of the two programs in tEmIc confirms these considerations and the witness for the left program indeed first sets `r3` to 1 and confirms this only in the last step setting `y` to 1.

7 Conclusions and Future Work

We have introduced a must/can analysis for the interpretation of knowledge-based programs inspired by the constructive semantics of synchronous programming languages. The resulting constructive interpretation provides lower and upper bounds for the possible executions. This interpretation has been shown to be monotone and to yield a least fixed point. We have also transformed knowledge-based programs to general rule systems with positive and negative premisses. Finally, we have described our tool TEMIC for constructive interpretation and temporal-epistemic model checking over CTLK and demonstrated some applications of interpreting knowledge-based programs including CTLK-guards.

Our epistemic logic could be complemented by group knowledge [14, Ch. 6], like common or distributed knowledge. The temporal dimension could be extended to "Linear-Time Logic" (LTL), and, more importantly, to include some notion of fairness. Criteria for ensuring decided least fixed points for the must/can interpretation beyond synchronicity would be desirable. Also a comparison with non-monotone inductive definitions [12], SOS rules with negative premisses [24], and solution strategies for epistemic specifications [5], would be of interest. On the other hand, the general constructive approach may be useful to complement existing intuitionistic approaches to the semantics of synchronous programming languages [22]. Finally, the domain of memory models should be covered more comprehensively by interpreting knowledge-based programs.

References

1. Aczel, P.: An introduction to inductive definitions. In: Barwise, J. (ed.) Handbook of Mathematical Logic, chap. C.7, pp. 783–818. North-Holland (1977)
2. Aspinall, D., Ševčík, J.: Java Memory Model examples: Good, bad and ugly. In: Proc. Verification and Analysis of Multi-Threaded Java-like Programs (VAMP 2007) (2007)
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
4. Baltag, A., Moss, L.S.: Logics for Epistemic Programs. Synth. **139**(2), 165–224 (2004). https://doi.org/10.1023/B:SYNT.0000024912.56773.5e
5. Baral, C., Gelfond, M.: Logic programming and knowledge representation. J. Logic Program. **19–20**(Suppl. 1), 73–148 (1994)
6. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Guernic, P.L., de Simone, R.: The synchronous languages twelve years later. Proc. IEEE **91**(1), 64–83 (2003)
7. Berry, G.: The foundations of Esterel. In: Plotkin, G., Stirling, C., Tofte, M. (eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner, pp. 425–454. Foundations of Computing Series, MIT Press (2000)
8. Berry, G.: The Constructive Semantics of Pure Esterel, Draft v3 (2002), https://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf
9. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press, 2nd edn. (2002)
10. de Haan, H.W., Hesselink, W.H., de Lavalette, G.R.R.: Knowledge-based asynchronous programming. Fund. Inform. **63**(2-3), 259–281 (2004)

11. Denecker, M., Bruynooghe, M., Marek, V.: Logic programming revisited: Logic programs as inductive definitions. ACM Trans. Comput. Logic **2**(4), 623–654 (2001)
12. Denecker, M., Ternovska, E.: A logic of nonmonotone inductive definitions. ACM Trans. Comput. Log. **9**(2), 14:1–14:52 (2008)
13. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Knowledge-based programs. Distr. Comput. **10**(4), 199–225 (1997)
14. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press (2003)
15. Fandinno, J., Faber, W., Gelfond, M.: Thirty years of epistemic specifications. Theo. Pract. Logic Program. **22**(6), 1043–1083 (2022)
16. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K.A. (eds.) Proc. 5th Intl. Conf. Symp. Logic Programming. pp. 1070–1080. MIT Press (1988)
17. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. Addison-Wesley, 3rd edn. (2005)
18. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language Lustre. Proc. IEEE **79**(9), 1305–1320 (1991)
19. Harper, R.: Practical Foundations of Programming Languages. Cambridge University Press (2013)
20. van der Hoek, W., Wooldridge, M.J.: Model checking knowledge and time. In: Bosnacki, D., Leue, S. (eds.) Proc. 9th Intl. Ws. Model Checking of Software (SPIN 2002). Lect. Notes Comp. Sci., vol. 2318, pp. 95–111. Springer (2002). https://doi.org/10.1007/3-540-46017-9_9
21. Lomuscio, A., Penczek, W.: Model checking temporal epistemic logic. In: van Ditmarsch et al. [29], chap. 8, pp. 397–441
22. Lüttgen, G., Mendler, M.: The intuitionism behind statecharts steps. ACM Trans. Comput. Log. **3**(1), 1–41 (2002). https://doi.org/10.1145/504077.504078
23. Manson, J., Pugh, W., Adve, S.: The Java memory model (2005), http://dl.dropbox.com/u/1011627/journal.pdf, draft
24. Mousavi, M., Phillips, I., Reniers, M.A., Ulidowski, I.: Semantics and expressiveness of ordered SOS. Inform. & Comput. **207**(2), 85–119 (2009). https://doi.org/10.1016/j.ic.2007.11.008
25. Pugh, W.: The Java memory model (1999–), http://www.cs.umd.edu/~pugh/java/memoryModel/
26. Sainsbury, R.M.: Paradoxes. Cambridge University Press, 3rd edn. (2009)
27. Su, K.: Model checking temporal logics of knowledge in distributed systems. In: McGuiness, D.L., Ferguson, G. (eds.) Proc. 19th Natl. Conf. Artificial Intelligence, 16th Conf. Innovative Applications of Artificial Intelligence (AAAI 2004). pp. 98–103. AAAI Press, MIT Press (2004)
28. Vahidi, A.: JDD: A pure Java BDD and Z-BDD library. https://bitbucket.org/vahidi/jdd (2003)
29. van Ditmarsch, H., Halpern, J.Y., van der Hoek, W., Kooi, B. (eds.): Handbook of Epistemic Logic. College Publ. (2015)
30. van Ditmarsch, H., Halpern, J.Y., van der Hoek, W., Kooi, B.: An introduction to logics of knowledge and belief. In: Handbook of Epistemic Logic [29], chap. 1, pp. 1–51
31. van Ditmarsch, H., van der Hoek, W., Kooi, B.: Dynamic Epistemic Logic, Synthese Library, vol. 337. Springer (2008)

Contextual Modal Type Theory with Polymorphic Contexts

Yuito Murase[1](✉), Yuichi Nishiwaki[2], and Atsushi Igarashi[1]

[1] Kyoto University, Kyoto, Japan
{murase@fos.kuis.kyoto-u.ac.jp, igarashi@kuis.kyoto-u.ac.jp}
[2] Tokyo, Japan
yuichi.nishiwaki@icloud.com

Abstract. Modal types—types that are derived from proof systems of modal logic—have been studied as theoretical foundations of metaprogramming, where program code is manipulated as first-class values. In modal type systems, modality corresponds to a type constructor for code types and controls free variables and their types in code values. Nanevski et al. have proposed *contextual modal type theory*, which has modal types with fine-grained information on free variables: modal types are explicitly indexed by *contexts*—the types of all free variables in code values.
This paper presents $\lambda_{\forall[]}$, a novel extension of contextual modal type theory with *parametric polymorphism over contexts*. Such an extension has been studied in the literature but, unlike earlier proposals, $\lambda_{\forall[]}$ is more general in that it allows multiple occurrence of context variables in a single context. We formalize $\lambda_{\forall[]}$ with its type system and operational semantics given by β-reduction and prove its basic properties including subject reduction, strong normalization, and confluence. Moreover, to demonstrate the expressive power of polymorphic contexts, we show a type-preserving embedding from a two-level fragment of Davies' $\lambda_{\bigcirc}$, which is based on linear-time temporal logic, to $\lambda_{\forall[]}$.

Keywords: Contextual modal types, Fitch-style modal lambda-calculi, Metaprogramming, Polymorphic contexts

1 Introduction

It is a common technique in metaprogramming to use code as a first-class value to generate, combine, and evaluate code at compile- and run-time. Type systems for first-class code are known to correspond to proof systems of modal logic under the Curry–Howard isomorphism [5,19,6,30,17]: Modality corresponds to a type constructor for code types, controlling free variables and their types in code values. Such modal type systems have been proposed for various areas of metaprogramming, including multi-stage computation [29,2,13], syntactic metaprogramming [7,27], and, more recently, applied to proof assistants [3,21,26].

Modal types come in two flavors: implicit and explicit contexts. On the one hand, modal types with implicit contexts do not show typing contexts—free variables and their types—of code values. A classical example of a modal type system with implicit contexts is $\lambda_{\bigcirc}$ [5], in which a code type is expressed by $\bigcirc T$ ("code of T"), no matter what variables are referenced in the code. It has been applied to real programming languages for multi-stage programming, such as MetaOCaml [2,13]. Since the type operator $\bigcirc$ is derived from the modality "next" in linear-time temporal logic, we call these code types linear-time temporal types. On the other hand, modal types with explicit contexts show typing contexts in code types. For example, the type of code `x+2` is expressed by $[x : \mathit{int}]\mathit{int}$, which stands for code of an integer expression that includes free occurrences of an integer variable x. Such types are often called contextual modal types [17]. Prior work points out that contextual modal types have advantages over linear-time temporal types in dealing with mutable reference cells and run-time code evaluation [12,24,14] although it is not actively applied to real multi-stage programming languages so far. Contextual modal types is rather known for its applications to proof assistants [20,3,21,26], where users can operate on code representation of proof terms with explicit contexts.

Some previous work [12,16,3,21,23] on contextual modal types has suggested *polymorphic contexts*—polymorphism over typing contexts in contextual modal types—to abstract part of typing contexts by context variables γ: For example, a type $\forall\gamma.[\gamma]T_1 \to [\gamma]T_2$ denotes functions that take code of type T_1 under an arbitrary typing context γ and return code of type T_2 under the same typing context γ. Although we can see that polymorphic contexts will play an important role in metaprogramming with contextual modal types, its type-theoretic foundations are not fully investigated yet.

Our contributions. This paper proposes a novel contextual modal type theory $\lambda_{\forall[]}$ that provides a type-theoretic foundation for polymorphic contexts. Our technical contributions are summarized below:

– We develop contextual modal type theory $\lambda_{\forall[]}$ with polymorphic contexts formally: we give its syntax, type system, and operational semantics given by β-reduction. A notable feature of $\lambda_{\forall[]}$ is that it allows multiple occurrences of context variables in a single context, e.g., $\forall\gamma_1.\forall\gamma_2.[\gamma_1, x : T_1, \gamma_2]T_2$.
– We prove basic properties of $\lambda_{\forall[]}$: subject reduction, strong normalization, and confluence. Our strong normalization proof is based on Girard's parametric reducibility method, which is adapted to polymorphic contexts.
– To demonstrate the expressive power of polymorphic contexts, we give translation from a two-level fragment of $\lambda_{\bigcirc}$ [5] to $\lambda_{\forall[]}$ and prove that the translation preserves typing. To our knowledge, this is the first result that formally describes the relation between linear-time temporal types and contextual modal types. We will see that $\lambda_{\forall[]}$'s major advantage that allows multiple occurrences of context variables in a single context plays a vital role.

Organization of the paper. Section 2 provides motivating examples from metaprogramming. Our formal development starts with a simple Fitch-style modal type theory $\lambda_{[]}$ in Section 3. We extend $\lambda_{[]}$ to $\lambda_{\forall[]}$ with polymorphic contexts and prove subject reduction in Section 4; we prove strong normalization of $\lambda_{\forall[]}$ in Section 5. Section 6 develops a sound embedding from linear-time temporal types to contextual modal types. Finally, we discuss related work in Section 7 and give a conclusion in Section 8.

2 Motivation

This section provides examples from common metaprogramming use cases. We use a hypothetical OCaml-like language with contextual modal types we present later. Note that the language is supposed only to illustrate the type theory's informal ideas and is not intended as practical language.

2.1 Simple Contextual Modal Types: Specializing Power Function

First, we show a typical example from staged computation, the power function, to demonstrate how we can use contextual modal types for staged computation.

```
(* val pow : int -> [int |- int] *)
let rec pow n = match n with
  | 0 -> ‘<x: int> 1
  | n -> let u = pow (n-1) in ‘<x: int>(x * ,1(u)[x])

(* val power4 : int -> int *)
let power4 = ,0(‘<>(fun x:int -> ,1(pow 4)[x]))[]
```

The function `pow` generates a piece of code: `x * (... * (x * 1)...)` that multiplies variable `x` `n` times; the function `power4` puts the code generated by `pow` under function abstraction and evaluates the code at run-time to obtain a function value to compute x^4 without recursion.

This example uses two constructs for code manipulation: *quote* of the form ‘<Γ>M and *unquote* of the form ,n(M)[$M_1, \dots, M_k$]. The former, which is similar to quasi-quotation in Lisp, generates code of an expression M paired with a variable environment Γ under which the code is evaluated. In the example, the quote `‘<x: int> 1` is code of constant `1` with the environment with single integer variable `x`. The quote has a contextual modal type `[int |- int]`, where the premise (`int` on the left of `|-`) corresponds to the environment `x:int` and the succedent (`int` on the right) to the code body.

Given a contextual modal type $[C \vdash T]$, we call C a *context*. A context is a sequence of types and does not involve variables. Similarly to de Bruijn indices, we identify variables in a context by their position rather than by their names. For instance, two quotes, `‘<x:int, y:int>x` and `‘<z:int, w:int>z`, are considered α-equivalent because both use the first variable in the environment even though the variable names in the two environments are different. Both terms have the same type `[int, int |- int]`.

An unquote ,n(M)[$M_1, \dots, M_k$] is used to expand a code value M. For example, `,1(u)[x]` expands `u` of type `[int |- int]`. In addition to the code to be expanded, an unquote involves two annotations, an *explicit substitution* $[M_1, \ldots, M_k]$ and a *stage transition* n. An explicit substitution provides the definitions of the variables in the environment of a quote value. In the example code, `,1(u)[x]` supplies an explicit substitution `[x]` as the definition for a single-variable context `int`. If `u` is `‘<y:int>y * 1`, then the unquote will expand to `x * 1`, replacing `y` with its definition `x`. Roughly speaking, a stage transition represents the number of nested quotes surrounding M. The expression `,1(u)[x]` applies the explicit substitution to `u`, and splices the obtained code into the surrounding quote. Thus, the code `‘<x: int>(x * ,1(u)[x])` adds "`x *`" to the code denoted by `u`. On the contrary, the unquote `,0(‘<>(fun x:int -> ,1(pow 4)[x]))[]` computes `‘<>(fun x:int -> ,1(pow 4)[x])` (to obtain the code value `fun x:int -> (x * (x * (x * (x * 1))))` with the empty environment) and expands it; since there is no surrounding quote, the expansion amounts to running the code. In this sense, the unquote in this language can be considered as unquote in Lisp-like languages if the stage transition is 1 and as `eval` function if it is 0.

2.2 Polymorphic Contexts: Macro `repeat`

Secondly, consider a macro called `repeat`, which repeats a given piece of code n times. For example, we expect Lisp code `(repeat 2 (print "hello"))` to show `hello` two times. We can imitate such a macro as follows:

```
(* val repeat : int -> [string -> unit |- unit]
                    -> [string -> unit |- unit]  *)
let rec repeat n body = match n with
  | 0 -> ‘<pr: string -> unit>(())
  | n -> let u = repeat (n-1) body in
         ‘<pr: string -> unit>(,1(u)[pr]; ,1(body)[pr])
```

This function `repeat` takes an integer `n` for the number of repetitions and code to be repeated. For example, a macro call in Lisp `(repeat 2 (print "hello"))` can be represented below.

```
,1(repeat 2 ‘<pr:string -> unit>(pr "hello"))[print]
```

To model macro expansion, we assume the whole code with macro calls is surrounded by a quote; hence, we use the stage transition 1, instead of 0, to splice the result of the macro call of `repeat`. Note that the environment `pr:string -> unit` is expected to be the function `print`. After applying the function `repeat`, we obtain the following code.

```
,1(‘<pr:string -> unit>(pr "hello"; pr "hello"; ()))[print]
```

Finally, by evaluating unquote, the code is fully expanded (with substituting library the function `print` for `pr`) to

```
print "hello"; print "hello"; () .
```

A problem with the function `repeat` is that it accepts code values with an environment that consists of a single variable of type `string -> unit`. We rather expect the function to accept code values with various patterns of contexts and to have multiple types that differ only in contexts: e.g.,

- `int -> [string -> unit |- unit] -> [string -> unit |- unit]`,
- `int -> [string -> unit, int, int |- unit]`
  `-> [string -> unit, int, int |- unit]`, and
- `int -> [unit -> unit |- unit] -> [unit -> unit |- unit]` .

We will resolve this issue by abstracting the context part of the function with a *context variable* `G`. As a result, we obtain the type for generic `repeat`: `forall G. int -> [G |- int] -> [G |- int]`. We call the type starting with `forall G.` a *polymorphic context type*, which means that we can instantiate the context variable `G` with any context. We can implement this generic function `poly_rep` by using a context variable as follows.

```
(* val poly_rep : forall G. int -> [G |- unit] -> [G |- unit] *)
let rec poly_rep [G] n body =
  match n with
  | 0 -> `<xs: G>(())
  | n -> let u = poly_rep [G] (n-1) body in
         `<xs: G>(,1(u)[xs]; ,1(body)[xs])
```

This function takes an additional context argument `G`, which is used in quotes. `xs` is a *series variable*, which is a novel sort of variables in this paper. A series variable stands for a sequence of (ordinary) variables—corresponding to the fact that a context variable stands for a sequence of types—and forms an environment by pairing with a context variable. For example, `xs:G` will represent environment `x:int, y:string` if we substitute `x, y` for `xs`, and `int, string` for `G`. We can also use series variables for explicit substitution. If we use a series variable in an explicit substitution, as in `,1(u)[xs]`, `xs` stands for an explicit substitution consists of a series of variables. For instance, if `xs:G` expands to `x:int, y:string`, then `,1(u)[xs]` also expands to `,1(u)[x,y]`. In this case, series variables work like identity substitutions in prior work [26,3,21,23], which pass variables from an environment to explicit substitutions as-is.

Using `poly_rep`, we can repeat code with two variables as follows:

```
poly_rep [unit->int, int->unit] 3
  (`<rand:unit->int, printInt:int->unit>(printInt(rand())))
```

We apply to the context `unit->int, int->unit` in order to instantiate the context variable `G`. It is worth noting that the series variables accompanied by `G` will also be replaced automatically with fresh variables. In this case, the quote `` `<xs: G>(,u[xs]; ,body[xs]) `` will turn into

```
`<x: unit->int, y:int->unit>(,u[x,y]; ,body[x,y])
```

where the series variable `xs` is replaced with fresh variables `x,y`. This way, a mapping between variables and types is well maintained.

2.3 More Polymorphic Contexts: Combining Different Environments

Sometimes, we might want to use pieces of code with different environments. Consider a function `generic_plus`, which takes two pieces of code as arguments and returns a piece of code that sums the values of the two arguments. We can implement such a function with ease.

```
(* val generic_plus:
        forall G H. [G |- int] -> [H |- int] -> [G, H |- int] *)
let generic_plus [G H] x y = ‘<xs:G, ys:H>(,1(x)[xs] + ,1(y)[ys])
```

It takes two context variables `G` and `H` and puts them together in the same context. As a result, we can use variables from both contexts. Although this example is very simple, it demonstrates the novel feature of our contextual modal type theory: it permits multiple occurrences of context variables in the same context, as in `[G, H |- int]`. As far as we understand, previous work that supports context polymorphism only allows a single occurrence of context variables. We discuss the detail in Section 7.

One may wonder whether multiple occurrences of context variables are useful. As we answer in Section 6, this novel feature is crucial to achieve the expressibility of the multi-stage programming languages in the literature.

3 Simple Fitch-Style Contextual Modal Type Theory

As an introduction to contextual modal types, this section formulates simple contextual type theory $\lambda_{[]}$ without polymorphic contexts. Nanevski et al. [17] formulated their original contextual modal type theory in dual-context style [19,6,11], which has judgments with two-level contexts. In contrast, we formulate $\lambda_{[]}$ in so-called Fitch- or Kripke-style [4,1,15,6,31]. We choose this design because the Fitch-style formulation provides Lisp-like quote/unquote syntax, which is akin to that in linear-temporal type theories [5,30], and hence it is easier to compare these two type theories. We demonstrate a formal comparison in Section 6.

We obtain $\lambda_{[]}$ by extending S4 Fitch-style modal calculus with contextual modal type theory. One can consider it a combination of the Fitch-style modal calculi by Valliappan et al. [31], and the contextual extension by Nanevski et al. [17]. At the same time, we tweak definitions for an extension to polymorphic contexts in Section 4.

3.1 Syntax and Type System

Types and terms in $\lambda_{[]}$ are shown in Fig. 1. Types consist of base types ranged over by ι, function types $S \to T$, and contextual modal types $[C \vdash T]$. A contextual modal type $[C \vdash T]$ generalizes an S4 modal type $\Box T$ by adding a *context* C, which is a finite sequence of types. It describes code of type T with free variables whose types are C. Note that a contextual modal type with a empty context $[\bullet \vdash T]$ has the same meaning as $\Box T$, which denotes closed code

Types	S, T	$::= \iota \mid S \to T \mid [C \vdash T]$
Contexts	C, D	$::= \bullet \mid C, T$
Stage transitions	k	$\in \mathbb{N}$
Terms	M, N	$::= x \mid \lambda x^T.M \mid MN \mid \mathsf{quo}\langle\hat{\Gamma}\rangle M \mid \mathsf{unq}_k M[\theta]$
Explicit Subst.	θ	$::= \bullet \mid \theta, M$
Named Contexts	Γ, Δ	$::= \bullet \mid \Gamma, x\colon T \mid \Gamma, \text{🔒}$
		($\hat{\Gamma}$ and $\hat{\Delta}$ denote named contexts with no 🔒.)

Fig. 1. Syntax of $\lambda_{[]}$

of type T. In addition to standard terms of simply typed lambda calculus, $\lambda_{[]}$ has two forms, quote $\mathsf{quo}\langle\hat{\Gamma}\rangle M$ and unquote $\mathsf{unq}_k M[\theta]$. We define stage transitions as natural numbers, and explicit substitutions as sequences of terms.

We often use the word *named contexts* for typing contexts with variables and use "contexts" for type-only ones. Similarly to other Fitch-style formulations, $\lambda_{[]}$ extends named contexts with a special symbol 🔒 (called lock) that delimits levels of variables. For example, in a named context $x\colon T_1, \text{🔒}, y\colon T_2, z\colon T_3$, the variable x has one higher level than y and z (we will revisit the notion of levels in the definition of free variables). A named context is well formed iff the variables in it do not have duplication; we assume that all named contexts are well formed. We also require a named context in a quote to be single-level, i.e., not to contain 🔒. We write $\hat{\Gamma}$ for such 🔒-free named contexts. $\mathsf{rg}(\hat{\Gamma})$ denotes the range of $\hat{\Gamma}$, a context obtained by forgetting variables in $\hat{\Gamma}$, and $\mathsf{dom}\,(\Gamma)$ denotes the domain of Γ, the set of variables in Γ (locks can appear in Γ, unlike rg). We also define the weakening relation $\Gamma_1 \leq \Gamma_2$ as follows.

$$\frac{}{\bullet \leq \bullet} \qquad \frac{\Gamma_1 \leq \Gamma_2}{\Gamma_1, x\colon T \leq \Gamma_2, x\colon T} \qquad \frac{\Gamma_1 \leq \Gamma_2}{\Gamma_1 \leq \Gamma_2, x\colon T} \qquad \frac{\Gamma_1 \leq \Gamma_2}{\Gamma_1, \text{🔒} \leq \Gamma_2, \text{🔒}}$$

As is common in other Fitch-style formulations, $\lambda_{\forall[]}$ has a somewhat complex binding structure. We show the definition of free variables in Fig. 2. For a term M and integer k, $\mathsf{FV}_k(M)$ is a set of free variables in M at level k, which roughly stands for the number of quotes surrounding M. Since an unquote $\mathsf{unq}_{k_1} M[\theta]$ cancels k_1 surrounding quotes, the level is lowered by k_1. $\lambda_{[]}$ has two binding forms: A lambda abstraction $\lambda x^T.M$ binds all level-0 free occurrences of x in M and a quote $\mathsf{quo}\langle\hat{\Gamma}\rangle M$ binds all level-0 free variables from $\hat{\Gamma}$ in M. According to these binding forms, we define α-equivalence (but omit its definition). For example, $\lambda x^{[T_1 \vdash T_2]}.\mathsf{quo}\langle x\colon T_1\rangle(\mathsf{unq}_1(x)[x])$ is α-equivalent to $\lambda y^{[T_1 \vdash T_2]}.\mathsf{quo}\langle z\colon T_3\rangle(\mathsf{unq}_1(z)[y])$. As we shall see later, the typing rules of $\lambda_{[]}$ enforce well-typed terms to be closed with regard to negative-level free variables. Thus, we only care about positive-level free variables in this paper and assume that the meta variable k ranges over natural numbers.

Typing rules are given in Fig. 3. The judgment $k\colon \Gamma \lhd \Delta$ states that there is a stage transition k between two named contexts Γ and Δ. The rules mean that k is the number of locks between Γ and Δ, e.g., $0\colon x\colon T \lhd x\colon T$ and $2\colon y\colon T_1 \lhd y\colon T_1, \text{🔒}, \text{🔒}, z\colon T_2$. The judgments $\Gamma \vdash M\colon T$ and $\Gamma \vdash \theta\colon C$ state

$\boxed{\mathsf{FV}_k(M)}$ $\boxed{\mathsf{FV}_k(\theta)}$

$$\mathsf{FV}_k(x) = \begin{cases} \{x\} & \text{if } k = 0 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{FV}_k(\lambda x^T.M) = \begin{cases} \mathsf{FV}_k(M) - \{x\} & \text{if } k = 0 \\ \mathsf{FV}_k(M) & \text{otherwise} \end{cases}$$

$$\mathsf{FV}_k(M\ N) = \mathsf{FV}_k(M) \cup \mathsf{FV}_k(N)$$

$$\mathsf{FV}_k(\mathsf{quo}\langle\hat{\Gamma}\rangle M) = \begin{cases} \mathsf{FV}_0(M) - \mathsf{dom}\,(\hat{\Gamma}) & \text{if } k = -1 \\ \mathsf{FV}_{k+1}(M) & \text{otherwise} \end{cases}$$

$$\mathsf{FV}_{k_2}(\mathsf{unq}_{k_1} M[\theta]) = \mathsf{FV}_{k_2-k_1}(M) \cup \mathsf{FV}_{k_2}(\theta)$$

$$\mathsf{FV}_k(\bullet) = \emptyset \qquad \mathsf{FV}_k(\theta, M) = \mathsf{FV}_k(\theta) \cup \mathsf{FV}_k(M)$$

Fig. 2. Free variables. This definition assumes k is an integer, but typing rules enforces that $\mathsf{FV}_k(M) = \emptyset$ if $k < 0$.

that term M has type T and explicit substitution θ has context C under named context Γ, respectively. The rules for variable x, lambda abstraction $\lambda x^T.M$, and application $M_1\ M_2$ are almost the same as those in simply typed lambda calculus, except that we only care about variables from $\mathsf{tail}\,(\Gamma)$, the level-0 part of Γ. The type of a quote $\mathsf{quo}\langle\hat{\Gamma}\rangle M$ is derived by popping all level-0 variables in the named context (Recall lock does not appear in $\hat{\Gamma}$). Thus, $\hat{\Gamma}$ binds all level-0 free variables in M. An unquote $\mathsf{unq}_k M[\theta]$ uses θ as a substitution for the context C, and k as the stage transitions between M and θ. We call a judgment *derivable* when it is derived from these typing rules. We assume that judgments in this paper are derivable if not stated explicitly.

3.2 Substitution

We define substitution on terms and explicit substitutions. We follow the style of Valliappan et al. [31], which proposes simultaneous substitution on all free variables with any level. We provide definitions related to substitutions in Fig. 4.

A substitution typing judgment $\vdash \sigma\colon \Delta \Rightarrow \Gamma$ denotes that we can replace a named context Δ with another Γ by applying a substitution σ, e.g., $\vdash (z := x\,y)\colon (z\colon T_2) \Rightarrow (x\colon T_1 \to T_2, y\colon T_1)$. A lock substitution $🔒_k$ has two roles. First, they provide information on the level of free variables to be substituted. For example, if $\sigma = \sigma_1, 🔒_k, \sigma_2$ where σ_2 does not have lock substitutions, σ_2 substitutes level-0 free variables, and σ_1 substitutes higher-level free variables. Second, they replace the lock themselves. If σ has a lock substitution $🔒_k$, it means that it replaces a lock in Δ with k locks in Γ.

$\boxed{k\colon \Gamma \lhd \Delta}$

$$\frac{}{0\colon \Gamma \lhd \Gamma} \qquad \frac{k\colon \Gamma \lhd \Delta}{k\colon \Gamma \lhd \Delta, x\colon T} \qquad \frac{k\colon \Gamma \lhd \Delta}{k+1\colon \Gamma \lhd \Delta, \text{🔒}}$$

$\boxed{\Gamma \vdash M\colon T}$ $\boxed{\Gamma \vdash \theta\colon C}$

$$\frac{x\colon T \in \mathsf{tail}\,(\Gamma)}{\Gamma \vdash x\colon T} \qquad \frac{\Gamma, x\colon T_1 \vdash M\colon T_2}{\Gamma \vdash \lambda x^{T_1}.M\colon T_1 \to T_2} \qquad \frac{\Gamma \vdash M_1\colon T_1 \to T_2 \qquad \Gamma \vdash M_2\colon T_1}{\Gamma \vdash M_1\, M_2\colon T_2}$$

$$\frac{\Gamma, \text{🔒}, \Delta \vdash M\colon T}{\Gamma \vdash \mathsf{quo}\langle\hat{\Delta}\rangle M\colon [\mathsf{rg}(\hat{\Delta}) \vdash T]} \qquad \frac{\Gamma \vdash M\colon [C \vdash T] \qquad \Delta \vdash \theta\colon C \qquad k\colon \Gamma \lhd \Delta}{\Delta \vdash \mathsf{unq}_k M[\theta]\colon T}$$

$$\frac{}{\Gamma \vdash \bullet\colon \bullet} \qquad \frac{\Gamma \vdash \theta\colon C \qquad \Gamma \vdash M\colon T}{\Gamma \vdash \theta, M\colon C, T}$$

Auxiliary function

$$\mathsf{tail}\,(\bullet) = \bullet \qquad \mathsf{tail}\,(\Gamma, \text{🔒}) = \bullet \qquad \mathsf{tail}\,(\Gamma, x\colon T) = \mathsf{tail}\,(\Gamma), x\colon T$$

Fig. 3. Typing rules of $\lambda_{[]}$

Substitution application on terms $M[\sigma]$ and explicit substitutions $\theta[\sigma]$ performs actual substitution operations. They are defined to satisfy the following lemma, which is expected by the intuition of substitution typing.

Lemma 1 (Substitution Lemma). *If $\Gamma \vdash M\colon T$ and $\vdash \sigma\colon \Gamma \Rightarrow \Delta$, then $\Delta \vdash M[\sigma]\colon T$.*

For example, let us consider $\Gamma \vdash (\mathsf{unq}_1(x)[y])\, y\colon T$, where $\Gamma = x\colon [S \vdash S \to T], \text{🔒}, y\colon S$. We can construct the following substitution that provides a term for each variable in Γ.

$$\vdash (x \coloneqq x', \text{🔒}_0, y \coloneqq z\, w)\colon \Gamma \Rightarrow (x'\colon [S \vdash S \to T], z\colon S \to S, w\colon S)$$

This substitution replaces level-0 occurrences of y to $z\, w$ and level-1 occurrences of x to x'. 🔒_0 in the substitution denotes that level-1 free variables of target terms are mapped to level-0 terms; that is why the level-0 term x' is supplied for the level-1 variable x. We can observe that the substitution is applied as follows.

$$((\mathsf{unq}_1(x)[y])\, y)[x \coloneqq x', \text{🔒}_0, y \coloneqq z\, w] \tag{1}$$
$$= (\mathsf{unq}_1(x)[y])[x \coloneqq x', \text{🔒}_0, y \coloneqq z\, w]\, (y[x \coloneqq x', \text{🔒}_0, y \coloneqq z\, w]) \tag{2}$$
$$= (\mathsf{unq}_0(x[x \coloneqq x'])[y[x \coloneqq x', \text{🔒}_0, y \coloneqq z\, w]])\, (y[x \coloneqq x', \text{🔒}_0, y \coloneqq z\, w]) \tag{3}$$
$$= (\mathsf{unq}_0(x')[z\, w])\, (z\, w) \tag{4}$$

The most interesting equation is the one from (2) to (3). The substitution for x is shifted by 1 level, and the stage transition of the unquote changes from 1 to 0 to align staging levels. The resulting term is given type T under the new named context, as the substitution lemma states.

Substitution $\sigma ::= \bullet \mid \sigma, x := M \mid \sigma, \text{🔒}_k$

$\boxed{\vdash \sigma\colon \Delta \Rightarrow \Gamma}$

$$\frac{}{\vdash \bullet\colon \bullet \Rightarrow \Gamma} \qquad \frac{\vdash \sigma\colon \Delta \Rightarrow \Gamma_1 \qquad k\colon \Gamma_1 \lhd \Gamma_2}{\vdash (\sigma, \text{🔒}_k)\colon (\Delta, \text{🔒}) \Rightarrow \Gamma_2} \qquad \frac{\vdash \sigma\colon \Delta \Rightarrow \Gamma \qquad \Gamma \vdash M\colon T}{\vdash (\sigma, x := M)\colon (\Delta, x\colon T) \Rightarrow \Gamma}$$

$\boxed{M[\sigma]}$ $\boxed{\theta[\sigma]}$

$$x[\sigma] = \begin{cases} M & \text{if } x := M \in \mathsf{tail}\,(\sigma) \\ x & \text{otherwise} \end{cases}$$

$$(\lambda x^T.M)[\sigma] = \lambda x^T.(M[\sigma]) \textbf{ where } x \notin \mathsf{dom}\,(\mathsf{tail}\,(\sigma)) \text{ and } x \notin \mathsf{FV}_0(\sigma)$$

$$(M\ N)[\sigma] = (M[\sigma])\,(N[\sigma])$$

$$(\mathsf{quo}\langle\hat{\Gamma}\rangle M)[\sigma] = \mathsf{quo}\langle\hat{\Gamma}\rangle(M[\sigma, \text{🔒}_1, id_{\hat{\Gamma}}])$$

$$(\mathsf{unq}_k M[\theta])[\sigma] = \mathsf{unq}_{(\mathsf{count}(k,\sigma))}(M[\sigma \uparrow k])[\theta[\sigma]]$$

$$\bullet[\sigma] = \bullet \qquad (\theta, M)[\sigma] = \theta[\sigma], M[\sigma]$$

Auxiliary functions

$$\mathsf{FV}_k(\sigma, x := M) = \mathsf{FV}_k(\sigma) \cup \mathsf{FV}_k(M) \quad \mathsf{FV}_{k_2}(\sigma, \text{🔒}_{k_1}) = \begin{cases} \mathsf{FV}_{k_2 - k_1}(\sigma) & \text{if } k_2 \geq k_1 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathsf{tail}\,(\sigma, x := M) &= \mathsf{tail}\,(\sigma), x := M \\ \mathsf{tail}\,(\sigma, \text{🔒}_k) &= \bullet \\ \mathsf{count}(0, \sigma) &= 0 \\ \mathsf{count}((k_1+1), \bullet) &= k_1 + 1 \\ \mathsf{count}((k+1), (\sigma, x := M)) &= \mathsf{count}(k+1, \sigma) \\ \mathsf{count}((k_1+1), (\sigma, \text{🔒}_{k_2})) &= \mathsf{count}(k_1, \sigma) + k_2 \end{aligned} \qquad \begin{aligned} id_\bullet &= \bullet \\ id_{\Gamma, x\colon T} &= id_\Gamma, x := x \\ id_{\Gamma, \text{🔒}} &= id_\Gamma, \text{🔒}_1 \\ \sigma \uparrow 0 &= \sigma \\ \bullet \uparrow (k+1) &= \bullet \\ (\sigma, x := M) \uparrow (k+1) &= \sigma \uparrow (k+1) \\ (\sigma, \text{🔒}_{k_1}) \uparrow (k_2+1) &= \sigma \uparrow k_2 \end{aligned}$$

Fig. 4. Substitution

In Fig. 4, we also define identity substitutions that satisfies $\vdash id_\Gamma\colon \Gamma \Rightarrow \Gamma$ for any Γ. We can confirm that id_Γ does not affect the result of substitution, as stated in the following lemma. We use this property to define reduction later.

Lemma 2. *$M[\sigma] = M[id_\Gamma, \sigma]$ for any Γ.*

3.3 Local Soundness/Completeness and Reduction

According to Pfenning and Davies [19], the introduction and elimination rules for a type constructor should satisfy local soundness and local completeness, which correspond to β-reduction and η-expansion, respectively. We confirm that contextual modal types meet those conditions and then define reduction rules.

Local soundness states that the elimination rule is not too strong. For the case of contextual modal types, we can witness it by the following local reduction where we obtain the derivation $\mathcal{D}'$ by application of the substitution $[id_\Gamma, \text{🔒}_k, \hat{\Delta} := \theta]$, which we obtain from $\mathcal{E}$ and $k \colon \Gamma \lhd \Gamma'$. Here, $\hat{\Delta} := \theta$ denotes a substitution that maps each variable in $\hat{\Delta}$ to each term in θ.

$$\dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma, \text{🔒}, \hat{\Delta} \vdash M \colon T}}{\Gamma \vdash \mathsf{quo}\langle\hat{\Delta}\rangle M \colon [C \vdash T]} \quad \overset{\mathcal{E}}{\Gamma' \vdash \theta \colon C} \quad k \colon \Gamma \lhd \Gamma'}{\Gamma' \vdash \mathsf{unq}_k(\mathsf{quo}\langle\hat{\Delta}\rangle M)[\theta] \colon T} \quad \Rightarrow \quad \overset{\mathcal{D}'}{\Gamma' \vdash M[id_\Gamma, \text{🔒}_k, \hat{\Delta} := \theta] \colon T}$$

Local completeness states that the elimination rule is sufficiently strong. We can confirm this condition by the following local expansion (we assume that $\mathsf{rg}(\hat{\Delta}) = C$).

$$\overset{\mathcal{D}}{\Gamma \vdash M \colon [C \vdash T]} \Rightarrow \quad \dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma \vdash M \colon [C \vdash T]} \quad \overset{\vdots}{\Gamma, \text{🔒}, \hat{\Delta} \vdash \mathsf{dom}(\hat{\Delta}) \colon C} \quad \overset{\vdots}{1 \colon \Gamma \lhd \Gamma, \text{🔒}, \hat{\Delta}}}{\Gamma, \text{🔒}, \hat{\Delta} \vdash \mathsf{unq}_1 M[\mathsf{dom}(\hat{\Delta})] \colon T}}{\Gamma \vdash \mathsf{quo}\langle\hat{\Delta}\rangle \mathsf{unq}_1 M[\mathsf{dom}(\hat{\Delta})] \colon [C \vdash T]}$$

These patterns provide base cases for β-reduction and η-expansion. This paper focuses on β-reduction, which we define as follows.

Definition 1 (β-reduction). *We inductively define full reduction relations on terms and explicit substitutions, $\to_\beta$. We show main rules other than congruence below. We also define $\to_\beta^*$ as the reflexive transitive closure of $\to_\beta$.*

$$\dfrac{}{(\lambda x^S.M)\, N \to_\beta M[x := N]} \qquad \dfrac{}{\mathsf{unq}_k(\mathsf{quo}\langle \overrightarrow{x} \colon C\rangle M)[\theta] \to_\beta M[\text{🔒}_k, \overrightarrow{x} := \theta]}$$

We safely omit identity substitutions found in these rules, thanks to Lemma 2. We do not dive into the basic properties of $\lambda_{[]}$ for now because we discuss those of its extension $\lambda_{\forall[]}$ in Sections 4 and 5.

4 Polymorphic Contexts

This section proposes a novel type theory $\lambda_{\forall[]}$ that extends $\lambda_{[]}$ with polymorphic contexts. We quickly go through an overview of its syntax and semantics, focusing on the differences from $\lambda_{[]}$. As examples in Section 2, the critical idea of $\lambda_{\forall[]}$ is the notion of series variables, which can be considered the term representation for context variables.

4.1 Syntax, Type System, and Substitution

We provide the syntax of $\lambda_{\forall[]}$ in Fig. 5. First, $\lambda_{\forall[]}$ has two additional sorts of variables: context variables γ, δ, standing for contexts, and series variables $\mathbb{x}, \mathbb{y}$, representing sequences of variables. $\lambda_{\forall[]}$ adds polymorphic context types of the

$$\begin{array}{llcl}
\textbf{Types} & S, T & ::= & \ldots \mid \forall\gamma.T \\
\textbf{Contexts} & C, D & ::= & \ldots \mid C, \gamma \\
\textbf{Terms} & M, N & ::= & \cdots \mid \Lambda\gamma.M \mid M@C \\
\textbf{Explicit Subst.} & \theta & ::= & \ldots \mid \theta, \varkappa \\
\textbf{Named Contexts} & \Gamma, \Delta & ::= & \ldots \mid \Gamma, \varkappa\colon \gamma
\end{array}$$

Fig. 5. Syntax of $\lambda_{\forall[]}$

form $\forall\gamma.T$, which binds γ in T. It represents the set of types obtained by substituting any context C for the context variable γ. Two kinds of terms $\Lambda\gamma.M$ and $M@C$ are added as introduction and elimination for polymorphic context types. We allow C to include polymorphic context types; thus, polymorphism in $\lambda_{\forall[]}$ is impredicative. The definition of contexts means that we can abstract any part of a context with context variables, e.g., $\forall\gamma_1.\forall\gamma_2.[\gamma_1, \iota, \gamma_2 \vdash \iota]$. Accordingly, series variables can appear in explicit substitutions, and a pair of a series variable and a context variable can appear in a named context. FV is updated to accommodate series variables but we omit the definition here.

It is worth noting that context variables are not subject to staging. This allows us to use the same context variable across levels—for example, the type $\forall\gamma.[\gamma \vdash [\gamma \vdash T]]$ binds both occurrences of γ although they are in different levels. The definition of free context variables, denoted by $\mathsf{FCV}(-)$, is straightforward and we omit it in this paper.

We give additional typing rules and defining clauses of substitutions in Fig. 6. We also extend the auxiliary functions such as tail to accommodate the new syntax but we omit their definitions. The introduction and elimination rules for polymorphic context types are similar to those for the polymorphic types in System F [8]. The definition of context substitution $T[\gamma := C]$ for types is straightforward and omitted. The other rule for explicit substitutions states that we can add $\varkappa\colon \gamma$ to an explicit substitution if it appears in the level-0 part of Γ. The point of the extension of substitution is that a series variable can only be replaced with another series variable, not an explicit substitution. With these extensions, we can confirm that the substitution lemma holds as expected.

4.2 Context Substitution

We also define substitution for context variables, which is the most non-trivial part of $\lambda_{\forall[]}$. To describe the core idea of context substitution, let us consider a term $\mathsf{quo}\langle\varkappa\colon \gamma\rangle(\mathsf{unq}_1 M[\varkappa])$. If we naively substitute a context T, δ for the context variable γ in this term, we would obtain $\mathsf{quo}\langle\varkappa\colon (T, \delta)\rangle(\mathsf{unq}_1 M[\varkappa])$, where $\varkappa\colon (T, \delta)$ is simply ill formed as a named context. Instead, we will take the following steps.

1. We check the occurrences of γ in the named context of the quote $\mathsf{quo}\langle\varkappa\colon \gamma\rangle(\mathsf{unq}_1 M[\varkappa])$, and collect series variables that are associated to γ. In this case, we have only $\varkappa$.
2. We generate a series of fresh variables to be substituted for $\varkappa$. Each variable corresponds to each element of the new context T, δ. Suppose we generate

$\boxed{\Gamma \vdash M \colon T}$ $\boxed{\Gamma \vdash \theta \colon C}$

$$\frac{\Gamma \vdash M \colon T \qquad \gamma \notin \mathsf{FCV}(\Gamma)}{\Gamma \vdash \Lambda\gamma.M \colon \forall\gamma.T} \qquad \frac{\Gamma \vdash M \colon \forall\gamma.T}{\Gamma \vdash M@C \colon T[\gamma := C]}$$

$$\frac{\Gamma \vdash \theta \colon C \qquad \mathbb{x} \colon \gamma \in \mathsf{tail}(\Gamma)}{\Gamma \vdash \theta, \mathbb{x} \colon C, \gamma}$$

Substitution $\sigma ::= \ldots \mid \sigma, \mathbb{x} := \mathbb{y}$

$\boxed{M[\sigma]}$ $\boxed{\theta[\sigma]}$

$$\ldots \qquad (\Lambda\gamma.M)[\sigma] = \Lambda\gamma.(M[\sigma]) \text{ if } \gamma \notin \mathsf{FCV}(\sigma) \qquad (M@C)[\sigma] = (M[\sigma])@C$$

$$\ldots \qquad (\theta, \mathbb{x})[\sigma] = \begin{cases} \theta[\sigma], \mathbb{y} & \text{if } \mathbb{x} := \mathbb{y} \in \mathsf{tail}(\sigma) \\ \theta[\sigma], \mathbb{x} & \text{else} \end{cases}$$

$\boxed{\vdash \sigma \colon \Delta \Rightarrow \Gamma}$

$$\frac{\vdash \sigma \colon \Delta \Rightarrow \Gamma \qquad \mathbb{y} \colon \gamma \in \mathsf{tail}(\Gamma) \qquad \mathbb{x} \notin \mathsf{dom}(\Delta)}{\vdash \sigma, \mathbb{x} := \mathbb{y} \colon \Delta, \mathbb{x} \colon \gamma \Rightarrow \Gamma}$$

Fig. 6. Additional typing rules and definitions of substitutions in $\lambda_{\forall[]}$

new variables $x, \mathbb{y}$ for T, δ. As a result, we get a *variable series substitution* $\mathbb{x} := x, \mathbb{y}$.

3. We apply context substitution $\gamma := T, \delta$ to the named context $\mathbb{x} \colon \gamma$ along with $\mathbb{x} := x, \mathbb{y}$. As a result, we get a new named context $x \colon T, \mathbb{y} \colon \delta$.
4. We also apply the variable series substitution to $\mathsf{unq}_1 M[\mathbb{x}]$ and obtain $\mathsf{unq}_1 M[x, \mathbb{y}]$.
5. As a result, we obtain a substituted term $\mathsf{quo}\langle x \colon T, \mathbb{y} \colon \delta\rangle(\mathsf{unq}_1 M[x, \mathbb{y}])$.

In this way, substitution for context variables essentially requires three operations (1) to replace context variables with contexts, (2) to generate fresh variables to be substituted for series variables, and (3) to replace series variables with sequences of variables. We start its formal definition with the following new objects. We write G_v and G_s for infinite sequences of ordinary variables and series variables without duplication, respectively.

Context substitution	Σ	$::= \bullet \mid \Sigma, \gamma := C$
Variable series	$\overrightarrow{x}, \overrightarrow{y}$	$::= \bullet \mid \overrightarrow{x}, y \mid \overrightarrow{x}, \mathbb{y}$
Variable series substitution	$\bar{\sigma}$	$::= \bullet \mid \bar{\sigma}, \mathbb{x} := \overrightarrow{y} \mid \bar{\sigma}, 🔒$
Variable generator	G	$::= (G_v, G_s)$

A context substitution Σ maps context variables to contexts, and a variable series substitution $\bar{\sigma}$ maps series variables to variable series, that is, sequences of ordinary/series variables. Note that series substitution does not affect stage levels; hence, locks in series substitution are not annotated with stage transitions. A variable generator consists of streams of non-duplicating variables and series variables. We use it to generate fresh variables. $\mathsf{rg}\,(\bar{\sigma})$ denotes the variable series obtained from the range of $\bar{\sigma}$.

We define application of context substitution in Fig. 7. Application of a context substitution to types $T[\Sigma]$ and contexts $C[\Sigma]$ is straightforward; we simply replace context variables in a capture-avoiding manner. We omit their definitions from the figure. On the contrary, context substitution on terms $M[\Sigma;\bar{\sigma}]_G$ and explicit substitutions $\theta[\Sigma;\bar{\sigma}]_G$ comes with not only Σ but also a variable series substitution $\bar{\sigma}$ and a variable generator G. Σ is used to replace context variables in types in λ-abstractions and Γ in a quote; $\bar{\sigma}$ is used to substitute series variables in explicit substitutions and Γ in a quote. The most interesting is the case for a quote $\mathsf{quo}\langle\hat{\Gamma}\rangle M$: first, a variable series substitution $\bar{\sigma}'$ is generated by the auxiliary function $\mathsf{destruct}$ (Step 2 above); second, Σ and the generated $\bar{\sigma}'$ are applied to $\hat{\Gamma}$ to yield the new named context (Step 3); finally, we apply Σ and $\bar{\sigma}, \text{🔒}, \bar{\sigma}'$ to the body of the quote (Step 4), after removing variables in $\mathsf{dom}\,(\hat{\Gamma})$ and generated ones from the generator; here, $(G_v, G_s) - S$ means $(G_v \setminus S, G_s \setminus S)$. The auxiliary function $\mathsf{destruct}_G(\Gamma, \Sigma)$ scans Γ to find context variables in the domain of Σ, generates fresh (ordinary/series) variables by using $\mathsf{gensyms}$, and returns a variable series substitution. $\mathsf{gensyms}_G(C, V)$ produces a sequence of ordinary/series variables of the same length as C; fresh variables are chosen from earlier ones in G but not in V.

For example, consider applying $\Sigma = \gamma := T_1, \gamma'$ and the empty variable series substitution to $M = \mathsf{quo}\langle \mathbb{x}\colon \gamma, x\colon \iota, \mathbb{y}\colon \gamma\rangle M_0$. $\mathsf{destruct}_G((\mathbb{x}\colon \gamma, x\colon \iota, \mathbb{y}\colon \gamma), (\gamma := T_1, \gamma'))$ returns $\mathbb{x} := (x', \mathbb{x}'), \mathbb{y} := (y', \mathbb{y}')$ for some fresh $x', \mathbb{x}', y'$, and $\mathbb{y}'$ (with respect to G) and, thus, $M[\Sigma;\bullet]_G$ is $\mathsf{quo}\langle x'\colon T_1, \mathbb{x}'\colon \gamma', x\colon \iota, y'\colon T_1, \mathbb{y}'\colon \gamma'\rangle M_0'$ where $M_0' = M_0[\Sigma;(\bullet, \text{🔒}, \mathbb{x} := (x', \mathbb{x}'), \mathbb{y} := (y', \mathbb{y}'))]_{G'}$ and $G' = G - \{\mathbb{x}, x, \mathbb{y}, x', \mathbb{x}', y', \mathbb{y}'\}$.

We can confirm that context substitution preserves derivable judgments.

Lemma 3 (Context Substitution Lemma).

1. *If* $\Gamma \vdash M\colon T$ *then* $\Gamma[\Sigma;\bar{\sigma}] \vdash M[\Sigma;\bar{\sigma}]_{G'}\colon T[\Sigma]$ *where* $\bar{\sigma} = \mathsf{destruct}_G(\Gamma, \Sigma)$ *and* $G' = G - (\mathsf{dom}\,(\Gamma) \cup \mathsf{rg}\,(\bar{\sigma}))$ *for any* Σ *and* G.
2. *If* $\Gamma \vdash \theta\colon C$ *then* $\Gamma[\Sigma;\bar{\sigma}] \vdash \theta[\Sigma;\bar{\sigma}]_{G'}\colon C[\Sigma]$ *where* $\bar{\sigma} = \mathsf{destruct}_G(\Gamma, \Sigma)$ *and* $G' = G - (\mathsf{dom}\,(\Gamma) \cup \mathsf{rg}\,(\bar{\sigma}))$ *for any* Σ *and* G.

Although we use variable generators to get fresh variables, the result of context substitution should be equivalent under renaming. We can confirm this intuition by the following lemma.

Lemma 4. *If* $\Gamma \vdash M\colon T$, $\bar{\sigma}_1 = \mathsf{destruct}_{G_1}(\Gamma, \Sigma)$ *and* $\bar{\sigma}_2 = \mathsf{destruct}_{G_2}(\Gamma, \Sigma)$, *then there is a renaming substitution* σ *such that* $\Gamma[\Sigma;\bar{\sigma}_1] \vdash M[\Sigma;\bar{\sigma}_2]_{G_1'}[\sigma]\colon T[\Sigma]$ *with some* G_1'.

Corollary 1. *If* $\mathsf{dom}\,(\Sigma) \cap \mathsf{FCV}\,(\Gamma) = \emptyset$ *and* $\Gamma \vdash M\colon T$, *then* $M[\Sigma;\bullet]_{G_1} =_\alpha M[\Sigma;\bullet]_{G_2}$.

Based on this nature of context substitution, we may omit variable generators from context substitution applications.

$\boxed{M[\Sigma;\bar\sigma]_G}$

$$\begin{aligned}
x[\Sigma;\bar\sigma]_G &= x\\
(\lambda x^{T}.M)[\Sigma;\bar\sigma]_G &= \lambda x^{(T[\Sigma])}.(M[\Sigma;\bar\sigma]_G)\\
(M\;N)[\Sigma;\bar\sigma]_G &= (M[\Sigma;\bar\sigma]_G)\,(N[\Sigma;\bar\sigma]_G)\\
(\mathsf{quo}\langle\hat\Gamma\rangle M)[\Sigma;\bar\sigma]_G &= \mathsf{quo}\langle\hat\Gamma[\Sigma;\bar\sigma']\rangle(M[\Sigma;(\bar\sigma,\text{🔒},\bar\sigma')]_{G'})\\
&\quad\text{where } \bar\sigma' = \mathsf{destruct}_G(\hat\Gamma,\Sigma)\\
&\quad\text{and } G' = G - (\mathsf{dom}\,(\hat\Gamma)\cup\mathsf{rg}\,(\bar\sigma'))\\
(\mathsf{unq}_k M[\theta])[\Sigma;\bar\sigma]_G &= \mathsf{unq}_k(M[\Sigma;\bar\sigma\uparrow k]_G)[\theta[\Sigma;\bar\sigma]_G]\\
(\Lambda\gamma.M)[\Sigma;\bar\sigma]_G &= \Lambda\gamma.(M[\Sigma;\bar\sigma]_G) \qquad \text{if } \gamma\notin\mathsf{dom}\,(\Sigma) \text{ and } \gamma\notin\mathsf{FCV}\,(\Sigma)\\
(M@C)[\Sigma;\bar\sigma]_G &= (M[\Sigma;\bar\sigma]_G)@(C[\Sigma])
\end{aligned}$$

$\boxed{\theta[\Sigma;\bar\sigma]_G}$

$$\begin{aligned}
\bullet[\Sigma;\bar\sigma]_G &= \bullet\\
(\theta,M)[\Sigma;\bar\sigma]_G &= (\theta[\Sigma;\bar\sigma]_G),(M[\Sigma;\bar\sigma]_G)\\
(\theta,\cancel{x})[\Sigma;\bar\sigma]_G &= \begin{cases}(\theta[\Sigma;\bar\sigma]_G),\overrightarrow{y}\\ \quad\text{if } \cancel{x}\coloneqq\overrightarrow{y}\in\mathsf{tail}\,(\bar\sigma)\\ (\theta[\Sigma;\bar\sigma]_G),\cancel{x} \quad \text{otherwise}\end{cases}
\end{aligned}$$

$\boxed{\Gamma[\Sigma;\bar\sigma]}$

$$\begin{aligned}
\bullet[\Sigma;\bar\sigma] &= \bullet\\
(\Gamma,x\colon T)[\Sigma;\bar\sigma] &= \Gamma[\Sigma;\bar\sigma],x\colon T[\Sigma]\\
(\Gamma,\cancel{x}\colon\gamma)[\Sigma;\bar\sigma] &= \begin{cases}\Gamma[\Sigma;\bar\sigma],\overrightarrow{y}\colon C\\ \quad\text{if } \cancel{x}\coloneqq\overrightarrow{y}\in\mathsf{tail}\,(\bar\sigma)\\ \quad\text{and } \gamma\coloneqq C\in\Sigma\\ \Gamma[\Sigma;\bar\sigma],\cancel{x}\colon\gamma \quad \text{else}\end{cases}\\
(\Gamma,\text{🔒})[\Sigma;\bar\sigma] &= \Gamma[\Sigma;\bar\sigma\uparrow 1],\text{🔒}
\end{aligned}$$

Auxiliary functions

$$\begin{aligned}
\mathsf{destruct}_G((\Gamma,x\colon T),\Sigma) &= \mathsf{destruct}_G(\Gamma,\Sigma)\\
\mathsf{destruct}_G((\Gamma,\cancel{x}\colon\gamma),\Sigma) &= \begin{cases}\bar\sigma,\cancel{x}\coloneqq\overrightarrow{x} \qquad \text{if } \gamma\coloneqq C\in\Sigma\\ \quad\text{where } \bar\sigma=\mathsf{destruct}_G(\Gamma,\Sigma)\\ \quad\text{and } \overrightarrow{x}=\mathsf{gensyms}_G(C,\mathsf{dom}\,(\Gamma)\cup\mathsf{rg}\,(\bar\sigma))\\ \mathsf{destruct}_G(\Gamma,\Sigma) \quad \text{otherwise}\end{cases}\\
\mathsf{destruct}_G((\Gamma,\text{🔒}),\Sigma) &= \mathsf{destruct}_G(\Gamma,\Sigma),\text{🔒}\\
\mathsf{gensyms}_{(G_v,G_s)}(\bullet,V) &= \bullet\\
\mathsf{gensyms}_{(G_v,G_s)}((C,T),V) &= \mathsf{gensyms}_{(G_v,G_s)}(C,V\cup\{x\}),x\\
&\quad\text{where } x \text{ is the first element of } G_v \text{ such that } x\notin V\\
\mathsf{gensyms}_{(G_v,G_s)}((C,\gamma),V) &= \mathsf{gensyms}_{(G_v,G_s)}(C,V\cup\{\cancel{x}\}),\cancel{x}\\
&\quad\text{where } \cancel{x} \text{ is the first element of } G_s \text{ such that } \cancel{x}\notin V
\end{aligned}$$

Fig. 7. Context substitutions and variable series substitutions

4.3 Local Soundness and Completeness

Local soundness and local completeness are extended to polymorphic context types as follows. We use context substitution to obtain $\mathcal{D}'$ in the local reduction pattern. In this pattern, we observe $\mathsf{destruct}(\Gamma, \gamma := C) = \bullet$ because $\gamma \notin \mathsf{FCV}(\Gamma)$, and hence we get $\Gamma \vdash M[\gamma := C; \bullet] \colon T[\gamma := C]$. For the local expansion pattern, we have to pick a context variable δ that is fresh against Γ.

Local Soundness

$$\dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma \vdash M \colon T} \qquad \gamma \notin \mathsf{FCV}(\Gamma)}{\Gamma \vdash \Lambda\gamma.M \colon \forall\gamma.T}}{\Gamma \vdash (\Lambda\gamma.M)@C \colon T[\gamma := C]} \implies \overset{\mathcal{D}'}{\Gamma \vdash M[\gamma := C; \bullet] \colon T[\gamma := C]}$$

Local Completeness

$$\overset{\mathcal{D}}{\Gamma \vdash M \colon \forall\gamma.T} \implies \dfrac{\dfrac{\overset{\mathcal{D}'}{\Gamma \vdash M \colon \forall\gamma.T}}{\Gamma \vdash M@\delta \colon T[\gamma := \delta]} \qquad \delta \notin \mathsf{FCV}(\Gamma)}{\Gamma \vdash \Lambda\delta.(M@\delta) \colon \forall\delta.(T[\gamma := \delta])}$$

As a result, we obtain an additional reduction rule for $\to_\beta$ below.

$$\dfrac{}{(\Lambda\gamma.M)@C \to_\beta M[\gamma := C; \bullet]}$$

By using the substitution and context substitution lemmas, it is not hard to show subject reduction with regard to this β-reduction.

Theorem 1 (Subject Reduction).

1. *If $\Gamma \vdash M \colon T$ and $M \to_\beta M'$, then $\Gamma \vdash M' \colon T$.*
2. *If $\Gamma \vdash \theta \colon C$ and $\theta \to_\beta \theta'$, then $\Gamma \vdash \theta' \colon C$.*

Furthermore, β-reduction satisfies strong normalization and confluence. We only refer to confluence here because we will prove strong normalization in the next section.

Theorem 2 (Confluence). *If $\Gamma \vdash M \colon T$, $M \to_\beta^* N_1$ and $M \to_\beta^* N_2$, then there exists a term N_3 such that $N_1 \to_\beta^* N_3$ and $N_2 \to_\beta^* N_3$. The same holds also for well-typed explicit substitutions.*

Proof. We use Newmann's lemma [25]. We have strong normalizaiton from Theorem 3 (in Section 5) and weak confluence is easy to show.

5 Parametric Reducibility and Strong Normalization

This section provides a proof of strong normalization of β-reduction in $\lambda_{\forall[]}$. A common approach to proving strong normalization of a modal calculus is to provide a reduction-preserving translation to another strongly normalizing calculus such as simply typed lambda calculi [15,1]. We tried this approach, reducing strong normalization of $\lambda_{\forall[]}$ to that of System F [8]. However, it turned out not to be straightforward. Instead, we directly prove strong normalization of $\lambda_{\forall[]}$ using reducibility in this paper. We follow Girard's parametric reducibility [8] to define reducibility with polymorphic contexts. We also adopted techniques from logical relation for Fitch-style modal calculi proposed by Valliappan et al. [31] to extend reducibility to our Fitch-style modal type theory. Along with these existing methods, our approach requires several non-trivial extensions of reducibility for contextual modal types, which we detail in this section.

We start with the definition of neutral terms and explicit substitutions.

Definition 2 (Neutral Terms and Explicit Substitutions).

1. *A term M is* neutral *iff M is either of a variable, application, unquote, or context application.*
2. *An explicit substitution θ is* neutral *iff it can be derived from the rules below.*

$$\frac{}{\bullet \ \textit{is neutral}} \qquad \frac{\theta \ \textit{is neutral} \qquad M \ \textit{is neutral}}{\theta, M \ \textit{is neutral}} \qquad \frac{\theta \ \textit{is neutral}}{\theta, \lock \ \textit{is neutral}}$$

The definition of neutral terms is standard, while the one for neutral explicit substitutions is somewhat specific to $\lambda_{\forall[]}$ but straightforward: θ is neutral iff all terms in θ are neutral. Then, we define reducibility candidates.

Definition 3 (Reducibility Candidates). *Given a type T, let $\mathcal{R}$ be a set of derivable judgments of type T. We write $\mathcal{R}(\Gamma, M)$ iff $\Gamma \vdash M : T \in \mathcal{R}$. $\mathcal{R}$ is a* reducibility candidate of *T iff it satisfies all of the following properties.*

CR0 *If $\mathcal{R}(\Gamma, M)$ and $\Gamma \leq \Gamma'$, then $\mathcal{R}(\Gamma', M)$.*
CR1 *If $\mathcal{R}(\Gamma, M)$, then M is strongly normalizing with regard to $\rightarrow_\beta$.*
CR2 *If $\mathcal{R}(\Gamma, M)$ and $M \rightarrow_\beta M'$, then $\mathcal{R}(\Gamma, M')$.*
CR3 *If M is neutral, $\Gamma \vdash M : T$, and $\mathcal{R}(\Gamma, M')$ for all M' such that $M \rightarrow_\beta M'$, then $\mathcal{R}(\Gamma, M)$.*

We also define a reducibility candidate of context C similarly.

We abbreviate reducibility candidate as RC. As a next step, we define *reducibility candidate assignments* to define reducibility with parameters. We only need to care about reducibility candidates of contexts because $\lambda_{\forall[]}$ does not have polymorphic types.

RC assignment $\tilde{\Sigma} ::= \bullet \mid \tilde{\Sigma}, \gamma : C := \mathcal{R}$ (where $\mathcal{R}$ is an RC of C)

$\tilde{\Sigma}$ is well-formed if it does not have duplicating context variables in it. We assume that all reducibility candidate assignments are well-formed. We write $\mathsf{dom}\,(\tilde{\Sigma})$ for the set of context variables on the left side of $:=$ in $\tilde{\Sigma}$, and Σ for the context substitution that we can obtain by forgetting RCs in $\tilde{\Sigma}$.

On top of that, we define reducibility with parameters.

Definition 4 (Parametric Reducibility). *Given an RC assignment $\tilde{\Sigma}$, a type T, and a context C where $\mathsf{FCV}\,(T) \subseteq \mathsf{dom}\,(\tilde{\Sigma})$ and $\mathsf{FCV}\,(C) \subseteq \mathsf{dom}\,(\tilde{\Sigma})$, we define $\mathbf{Red}_T[\tilde{\Sigma}]$ and $\mathbf{Red}_C[\tilde{\Sigma}]$, a set of derivable judgments of a type $T[\Sigma]$ and a context $C[\Sigma]$, respectively, as follows. We write $\mathbf{Red}_T[\tilde{\Sigma}](\Gamma, M)$ iff $\Gamma \vdash M\colon T[\Sigma] \in \mathbf{Red}_T[\tilde{\Sigma}]$; similarly for $\mathbf{Red}_C[\tilde{\Sigma}](\Gamma, \theta)$.*

- *If $T = \iota$, then $\mathbf{Red}_T[\tilde{\Sigma}](\Gamma, M)$ iff M is strongly normalizing with regard to $\to_\beta$.*
- *If $T = T_1 \to T_2$, then $\mathbf{Red}_T[\tilde{\Sigma}](\Gamma, M)$ iff $\mathbf{Red}_{T_2}[\tilde{\Sigma}](\Delta, M\,N)$ for any Δ and N such that $\Gamma \le \Delta$ and $\mathbf{Red}_{T_1}[\tilde{\Sigma}](\Delta, N)$.*
- *If $T = [C \vdash T']$, then $\mathbf{Red}_T[\tilde{\Sigma}](\Gamma, M)$ iff $\mathbf{Red}_{T'}[\tilde{\Sigma}](\Delta', \mathsf{unq}_k M[\theta])$ for any Δ, Δ', k and θ such that $\Gamma \le \Delta$, $k\colon \Delta \lhd \Delta'$ and $\mathbf{Red}_C[\tilde{\Sigma}](\Delta', \theta)$.*
- *If $T = \forall\gamma.T'$, then $\mathbf{Red}_T[\tilde{\Sigma}](\Gamma, M)$ iff $\mathbf{Red}_{T'}[\tilde{\Sigma}, \gamma\colon C := \mathcal{R}](\Gamma, M@C)$ for any C and an RC $\mathcal{R}$ of C.*
- *If $C = \bullet$, then $\mathbf{Red}_C[\tilde{\Sigma}](\Gamma, \theta)$ always holds (where θ is always $\bullet$).*
- *If $C = C', T$, then $\mathbf{Red}_C[\tilde{\Sigma}](\Gamma, \theta)$ iff $\mathbf{Red}_{C'}[\tilde{\Sigma}](\Gamma, \theta')$ and $\mathbf{Red}_T[\tilde{\Sigma}](\Gamma, M)$ where $\theta = \theta', M$.*
- *If $C = C', \gamma$, then $\mathbf{Red}_C[\tilde{\Sigma}](\Gamma, \theta)$ iff $\mathbf{Red}_{C'}[\tilde{\Sigma}](\Gamma, \theta_1)$ and $\mathcal{R}(\Gamma, \theta_2)$ for some θ_1, θ_2, and $\mathcal{R}$ such that $\theta = \theta_1, \theta_2$ and $\gamma\colon D := \mathcal{R} \in \tilde{\Sigma}$.*

The definition for context variables is somewhat complicated. As $(C', \gamma)[\Sigma] = C'[\Sigma], D$, we need two reducible explicit substitutions θ_1 and θ_2 where θ_1 is for $C'[\Sigma]$ and θ_2 for D. Because D comes from the context variable γ, we use the RC $\mathcal{R}$ from $\tilde{\Sigma}$ to confirm that θ_2 is reducible.

The parametric reducibility is a reducibility candidate in fact, stated as the following lemma.

Lemma 5. *1. $\mathbf{Red}_T[\tilde{\Sigma}]$ is an RC of T.*
2. $\mathbf{Red}_C[\tilde{\Sigma}]$ is an RC of C.

We prove a few more auxiliary lemmas for the basic lemma. Firstly, we confirm that context substitution on types or context can be lifted to reducibility assignment.

Lemma 6. *1. $\mathbf{Red}_{T[\gamma:=C]}[\tilde{\Sigma}] = \mathbf{Red}_T[\tilde{\Sigma}, \gamma\colon C[\Sigma] := \mathbf{Red}_C[\tilde{\Sigma}]]$.*
2. $\mathbf{Red}_{D[\gamma:=C]}[\tilde{\Sigma}] = \mathbf{Red}_D[\tilde{\Sigma}, \gamma\colon C[\Sigma] := \mathbf{Red}_C[\tilde{\Sigma}]]$.

Besides, we state three lemmas that correspond to introduction of function types, contextual modal types, and polymorphic context types.

Lemma 7. *If $\Gamma, x\colon S[\Sigma] \vdash M\colon T[\Sigma]$ and $\mathbf{Red}_T[\tilde{\Sigma}](\Gamma', M[id_\Gamma, x := N])$ for any Γ' and N such that $\Gamma \le \Gamma'$ and $\mathbf{Red}_S[\tilde{\Sigma}](\Gamma', N)$, then $\mathbf{Red}_{S\to T}[\tilde{\Sigma}](\Gamma, \lambda x^S.M)$.*

Lemma 8. *If* $\Gamma, \text{🔒}, \overrightarrow{x}\colon C[\Sigma] \vdash M\colon T[\Sigma]$ *and* $\mathbf{Red}_T[\tilde{\Sigma}](\Gamma_2, M[id_{\Gamma_1}, \text{🔒}_k, \overrightarrow{x} := \theta])$ *for any* Γ_1*,* Γ_2*,* k *and* θ *such that* $\Gamma \leq \Gamma_1$*,* $k\colon \Gamma_1 \lhd \Gamma_2$ *and* $\mathbf{Red}_C[\tilde{\Sigma}](\Gamma_2, \theta)$*, then* $\mathbf{Red}_{[C \vdash T]}[\tilde{\Sigma}](\Gamma, \mathsf{quo}\langle \overrightarrow{x}\colon C[\Sigma]\rangle M)$*.*

Lemma 9. *If* $\Gamma \vdash M\colon T[\Sigma]$*,* $\gamma \notin \mathsf{FCV}(\Gamma) \cup \mathsf{FCV}(\Sigma) \cup \mathsf{dom}(\Sigma)$*, and* $\mathbf{Red}_T[\tilde{\Sigma}, \gamma\colon C := \mathcal{R}](\Gamma, M[\gamma := C; \bullet])$ *for any* C*,* $\mathcal{R}$ *such that* $\mathcal{R}$ *is an RC of* C*, then* $\mathbf{Red}_{\forall\gamma.T}[\tilde{\Sigma}](\Gamma, \Lambda\gamma.M)$*.*

We can prove these lemmas by CR3 and induction on the number of reduction steps of strongly normalizing terms/explicit substitutions.

Before the basic lemma, we define reducibility for named contexts. Although we would like something like $\mathbf{Red}_\Gamma[\tilde{\Sigma}]$, this definition does not work because it does not have information on how a named context with series variable $\mathbb{x}\colon \gamma$ will be replaced. Therefore we also need to pass series variables substitution, like $\mathbf{Red}_\Gamma[\tilde{\Sigma}, \bar{\sigma}]$ in the same way as context substitution for named contexts.

Definition 5 (Reducibility for Substitution). *Given an RC assignment* $\tilde{\Sigma}$*, a named context* Γ*, and a series substitution* $\bar{\sigma}$ *where* $\mathsf{FCV}(\Gamma) \subseteq \mathsf{dom}(\tilde{\Sigma})$*, we define* $\mathbf{Red}_\Gamma[\tilde{\Sigma}, \bar{\sigma}]$*, a set of derivable judgments of a named context* $\vdash \sigma\colon \Gamma[\Sigma; \bar{\sigma}] \Rightarrow \Delta$*, as follows. We write* $\mathbf{Red}_\Gamma[\tilde{\Sigma}, \bar{\sigma}](\Delta, \sigma)$ *iff* $\vdash \sigma\colon \Delta \Rightarrow \Gamma \in \mathbf{Red}_\Gamma[\tilde{\Sigma}, \bar{\sigma}]$*.*

- *If* $\Gamma = \bullet$*, then* $\mathbf{Red}_\Gamma[\tilde{\Sigma}, \bar{\sigma}](\Delta, \sigma)$ *always holds (where* $\sigma = \bullet$*).*
- *If* $\Gamma = \Gamma', x\colon T$*, then* $\mathbf{Red}_\Gamma[\tilde{\Sigma}, \bar{\sigma}](\Delta, \sigma)$ *iff* $\mathbf{Red}_{\Gamma'}[\tilde{\Sigma}, \bar{\sigma}](\Delta, \sigma')$ *and* $\mathbf{Red}_T[\tilde{\Sigma}](\Delta, M)$ *for some* σ'*,* M *such that* $\sigma = (\sigma', x := M)$*.*
- *If* $\Gamma = \Gamma', \mathbb{x}\colon \gamma$*, then* $\mathbf{Red}_\Gamma[\tilde{\Sigma}, \bar{\sigma}](\Delta, \sigma)$ *iff* $\mathbf{Red}_{\Gamma'}[\tilde{\Sigma}, \bar{\sigma}](\Delta, \sigma')$ *and* $\mathcal{R}(\Delta, \theta)$ *for some* σ'*,* θ *and* $\mathcal{R}$ *such that* $\gamma\colon C := \mathcal{R} \in \tilde{\Sigma}$*,* $\sigma = (\sigma', \overrightarrow{x} := \theta)$ *and* $\mathbb{x} := \overrightarrow{x} \in \bar{\sigma}$ *(*$\overrightarrow{x} := \theta$ *is a point-wise mapping between* $\overrightarrow{x}$ *and* θ*).*
- *If* $\Gamma = \Gamma', \text{🔒}$*, then* $\mathbf{Red}_\Gamma[\tilde{\Sigma}, \bar{\sigma}](\Delta, \sigma)$ *iff* $\mathbf{Red}_{\Gamma'}[\tilde{\Sigma}, \bar{\sigma}](\Delta \uparrow k, \sigma')$ *for some* σ' *and* k *such that* $\sigma = (\sigma', \text{🔒}_k)$*.*

We use series variables substitution in the third rule to generate a substitution for $(\mathbb{x}\colon \gamma)[\Sigma; \bar{\sigma}] = \overrightarrow{x}\colon C$. Finally, we prove the basic lemma.

Lemma 10 (Basic Lemma).

- *If* $\Gamma \vdash M\colon T$ *and* $\mathbf{Red}_\Gamma[\tilde{\Sigma}, \bar{\sigma}](\Delta, \sigma')$ *where* $\bar{\sigma} = \mathsf{destruct}(\Gamma, \Sigma)$*, then* $\mathbf{Red}_T[\tilde{\Sigma}](\Delta, M[\Sigma; \bar{\sigma}][\sigma'])$*.*
- *If* $\Gamma \vdash \theta\colon C$ *and* $\mathbf{Red}_\Gamma[\tilde{\Sigma}, \bar{\sigma}](\Delta, \sigma')$ *where* $\bar{\sigma} = \mathsf{destruct}(\Gamma, \Sigma)$*, then* $\mathbf{Red}_C[\tilde{\Sigma}](\Delta, \theta[\Sigma; \bar{\sigma}][\sigma'])$*.*

Strong normalization is proved as a special case of the basic lemma, where we choose Σ, $\bar{\sigma}$ and σ' as identity substitutions respectively.

Theorem 3 (Strong Normalization). *If* $\Gamma \vdash M\colon T$*, then* M *is strongly normalizing with regard to* $\rightarrow_\beta$*.*

$$\begin{array}{lll}
\textbf{Level-0 Types} & T^0, S^0 & \coloneqq \iota \mid S^0 \to T^0 \mid \bigcirc T^1 \\
\textbf{Level-0 Terms} & M^0, N^0 & \coloneqq x \mid \lambda x^{T^0}.M^0 \mid M^0\, N^0 \mid \mathsf{quo} M^1 \\
\textbf{Level-1 Types} & T^1, S^1 & \coloneqq \iota \mid S^1 \to T^1 \\
\textbf{Level-1 Terms} & M^1, N^1 & \coloneqq x \mid \lambda x^{T^1}.M^1 \mid M^1\, N^1 \mid \mathsf{unq} M^0 \\
\textbf{Named Contexts} & \Gamma^\circ, \Delta^\circ & \coloneqq \cdot \mid \Gamma^\circ, x :^0 T^0 \mid \Gamma^\circ, x :^1 T^1
\end{array}$$

$\boxed{\Gamma^\circ \vdash_i M^i : T^i}$ $(i \in \{0,1\})$

$$\frac{x :^i T^i \in \Gamma^\circ}{\Gamma^\circ \vdash_i x : T^i} \qquad \frac{\Gamma^\circ, x :^i T_1^i \vdash_i M^i : T_2^i}{\Gamma^\circ \vdash_i \lambda x^{T_1^i}.M^i : T_1^i \to T_2^i} \qquad \frac{\Gamma^\circ \vdash_i M^i : T_1^i \to T_2^i \qquad \Gamma^\circ \vdash_i N^i : T_1^i}{\Gamma^\circ \vdash_i M^i\, N^i : T_2^i}$$

$$\frac{\Gamma^\circ \vdash_1 M^1 : T^1}{\Gamma^\circ \vdash_0 \mathsf{quo} M^1 : \bigcirc T^1} \qquad \frac{\Gamma^\circ \vdash_0 M^0 : \bigcirc T^1}{\Gamma^\circ \vdash_1 \mathsf{unq} M^0 : T^1}$$

Fig. 8. Syntax and typing rules of $\lambda_\bigcirc$ (two-level fragment)

6 Embedding Linear-Time Temporal Type Theory

In multi-stage computation, contextual modal types are known to overcome weak points of linear-time temporal types from $\lambda_\bigcirc$ by Davies [5], regarding type safety of mutable reference cells and/or run-time code evaluation [12,24,14]. However, simple contextual modal theories, such as $\lambda_{[]}$, are known to be less expressive than linear-time temporal types. That is why polymorphic contexts are explored in the literature, which will endow expressiveness to contextual modal types. Then it is natural to ask if polymorphic contexts are strong enough to express linear-time temporal types. This section proves that the answer is yes, by providing a sound translation from linear-time temporal types to $\lambda_{\forall[]}$. We first define a two-level fragment of $\lambda_\bigcirc$, as a source language to simplify our embedding (Fig. 8). We call the fragment itself $\lambda_\bigcirc$ later in this paper. Then, we discuss the core insights of our embedding from $\lambda_\bigcirc$ and give a formal definition of our embedding from $\lambda_\bigcirc$ to $\lambda_{\forall[]}$. We also prove its soundness—the embedding preserves typing—while a proof that it also preserves semantics is left for future work.

$\lambda_\bigcirc$ has two stages: level-1 is the future stage. We define types and terms for each level (and metavariables are indexed by 0 or 1). A temporal type $\bigcirc T^1$ denotes a code for the future-stage value of T^1. Unlike contextual modal types, temporal types do not show context explicitly. Instead, typing judgments hold future-stage named contexts that implicitly represent contexts of those code types. A type judgment $\Gamma^\circ \vdash_i M^i : T^i$ (where $i = 0, 1$) means typing at the stage i, where Γ° includes variables of both levels. $\lambda_\bigcirc$ also has syntax for quote and unquote as in $\lambda_{\forall[]}$ but they are not annotated with named contexts and explicit substitutions. Typing rules do little with named contexts.

These differences lead to the difference in binding structure. For example, consider a $\lambda_\bigcirc$-term $\lambda f^{\bigcirc T_1^1 \to \bigcirc T_2^1}.\mathsf{quo}(\lambda x^{T_1^1}.\mathsf{unq}(f\, \mathsf{quo} x))$. In this term, the outer lambda binds the level-0 occurrence of f and the inner lambda binds the level-1 occurrence of x, although **quo** and **unq** are placed between binders and variable references. To embed $\lambda_{\bigcirc}$ to $\lambda_{\forall[]}$, we have to emulate this behavior of $\lambda_{\bigcirc}$.

We design our embedding from $\lambda_{\bigcirc}$ to $\lambda_{\forall[]}$ based on the following insights. First of all, we naturally embed quote and unquote of $\lambda_{\bigcirc}$ to those of $\lambda_{\forall[]}$ (by recovering missing annotations). Secondly, we can recover a hidden context of code types in $\lambda_{\bigcirc}$ from the types of level-1 free variables. For example, in the judgment

$$x :^0 \bigcirc\text{int}, y :^1 \text{int} \vdash_0 \mathsf{quo}\, y\colon \bigcirc\text{int},$$

the context of the type $\bigcirc$int (of $\mathsf{quo}\, y$) should be int because the named context has a level-1 binding $y :^1$ int. As a result, $\bigcirc$int under $x :^0 \bigcirc\text{int}, y :^1 \text{int}$ is embedded into [int $\vdash$ int]. Thirdly, recovered contexts of code types sometimes need to be extended. Let us consider the following judgment:

$$\cdot \vdash_0 \lambda f^{\bigcirc\text{int}\to\bigcirc\text{str}}.\mathsf{quo}(\lambda x^{\text{int}}.\mathsf{unq}(f\,\mathsf{quo}\, x)) : (\bigcirc\text{int} \to \bigcirc\text{str}) \to \bigcirc(\text{int} \to \text{str}).$$

The hidden context of the f is empty, and hence the type of f should be $[\bullet \vdash \text{int}] \to [\bullet \vdash \text{str}]$. However, f is used inside the level-1 binder λx^{int}, and hence this use of f should be typed as $[\text{int} \vdash \text{str}] \to [\text{int} \vdash \text{str}]$. We need to extend the context of the code type as an abstraction under **quo** extends the level-1 context. Thus, the polymorphic context type $\forall\gamma.[\gamma \vdash \text{int}] \to [\gamma \vdash \text{str}]$ is more appropriate for f. In this way, polymorphic contexts allow us to extend the context of an argument of code type, according to where the argument is used.

The formal definition of our embedding is shown in Figure 9. Level-1 types are translated to $\lambda_{\forall[]}$ types in a straightforward manner; the translation of level-0 types carries a context, which is used to signify the context of code types. If it translates a function type, we introduce a polymorphic context type to the argument type so that we can extend the context of the type later. For example, $(\bigcirc\text{int} \to \bigcirc\text{str}) \to \bigcirc(\text{int} \to \text{str})$ translates to $(\forall\gamma.(\forall\delta.[\gamma, \delta \vdash \text{int}]) \to [\gamma \vdash \text{str}]) \to [\bullet \vdash \text{int} \to \text{str}]$ under an empty context.

Before discussing term translation, we introduce *intermediate named contexts* $\tilde{\Gamma}$, an intermediate representation of embedded named contexts. Their structure is similar to named contexts in $\lambda_{\bigcirc}$ while its elements are variables and types of $\lambda_{\forall[]}$. We write $|\tilde{\Gamma}|_0$ for the level-0 fragment of $\tilde{\Gamma}$ and $|\tilde{\Gamma}|_1$ for the level-1 fragment of $\tilde{\Gamma}$. The relation $\Gamma^\circ \rightsquigarrow \tilde{\Gamma}$ means that Γ° can be translated into $\tilde{\Gamma}$. The point is that Γ° can be translated into different intermediate named contexts. For example, the $\lambda_{\bigcirc}$ named context $x :^1 T^1, y :^0 \bigcirc S^1, z :^0 \bigcirc S^1$ can be translated to both $x :^1 [\![T^1]\!], y :^0 [[\![T^1]\!] \vdash [\![S^1]\!]], z :^0 [[\![T^1]\!] \vdash [\![S^1]\!]]$ and $x :^1 [\![T^1]\!], y :^0 [[\![T^1]\!] \vdash [\![S^1]\!]], \cancel{x} :^1 \gamma, z :^0 [[\![T^1]\!], \gamma \vdash [\![S^1]\!]]$ due to the last rule of $\rightsquigarrow$. We use this relation to prove the soundness theorem (Theorem 4) later.

Term embedding carries an intermediate named context for two purposes. Firstly, it is used to infer a named context and an explicit substitution for quote and unquote. Secondly, it is used to know a missing context that we need to extend when using level-0 variables. The level-1 types in a named context always translate to polymorphic context types so that we can extend their context when those variables are used. $\mathsf{diff}\,(x, \tilde{\Gamma})$ determines the missing context, defined as $\mathsf{diff}\,(x, (\tilde{\Gamma}, x :^0 T, \tilde{\Delta})) = \mathsf{rg}(|\tilde{\Delta}|_1)$ (or undefined otherwise).

$\boxed{[\![T^1]\!]}$ $\boxed{[\![M^1]\!]_{\tilde{\Gamma}}}$

$$
\begin{aligned}
[\![\iota]\!] &= \iota \\
[\![T_1^1 \to T_2^1]\!] &= [\![T_1^1]\!] \to [\![T_2^1]\!] \\
[\![x]\!]_{\tilde{\Gamma}} &= x \\
[\![\lambda x^{T^1}.M^1]\!]_{\tilde{\Gamma}} &= \lambda x^{[\![T^1]\!]}.[\![M^1]\!]_{\tilde{\Gamma}, x :^1 [\![T^1]\!]} \\
[\![M^1\, N^1]\!]_{\tilde{\Gamma}} &= [\![M^1]\!]_{\tilde{\Gamma}}\, [\![N^1]\!]_{\tilde{\Gamma}} \\
[\![\mathsf{unq} M^0]\!]_{\tilde{\Gamma}} &= \mathsf{unq}_1 [\![M^0]\!]_{\tilde{\Gamma}} [\mathsf{dom}(|\tilde{\Gamma}|_1)]
\end{aligned}
$$

$\boxed{[\![T^0]\!]_C}$ $\boxed{[\![M^0]\!]_{\tilde{\Gamma}}}$

$$
\begin{aligned}
[\![\iota]\!]_C &= \iota \\
[\![T_1^0 \to T_2^0]\!]_C &= (\forall \gamma.[\![T_1^0]\!]_{C,\gamma}) \to [\![T_2^0]\!]_C \\
&\quad \text{for fresh } \gamma \\
[\![\bigcirc T^1]\!]_C &= [C \vdash [\![T^1]\!]] \\
[\![x]\!]_{\tilde{\Gamma}} &= x @ \mathsf{diff}\,(x, \tilde{\Gamma}) \\
[\![\lambda x^{T^0}.M^0]\!]_{\tilde{\Gamma}} &= \lambda x^{T}.[\![M^0]\!]_{\tilde{\Gamma}, x :^0 T} \\
&\quad \text{where } T = \forall \gamma.[\![T^0]\!]_{\mathsf{rg}(|\tilde{\Gamma}|_1),\gamma} \\
&\quad \text{for fresh } \gamma \\
[\![M^0\, N^0]\!]_{\tilde{\Gamma}} &= [\![M^1]\!]_{\tilde{\Gamma}}\, (\Lambda \gamma.[\![N^1]\!]_{\tilde{\Gamma}, \not{x} :^1 \gamma}) \\
&\quad \text{for a fresh } \not{x} \text{ and } \gamma \\
[\![\mathsf{quo} M^1]\!]_{\tilde{\Gamma}} &= \mathsf{quo}\langle |\tilde{\Gamma}|_1 \rangle [\![M^1]\!]_{\tilde{\Gamma}}
\end{aligned}
$$

Intermediate Named Context $\tilde{\Gamma} \coloneqq \cdot \mid \tilde{\Gamma}, x :^0 T \mid \tilde{\Gamma}, x :^1 T \mid \tilde{\Gamma}, \not{x} :^1 \gamma$

$\boxed{\Gamma^\circ \rightsquigarrow \tilde{\Gamma}}$

$$
\frac{}{\cdot \rightsquigarrow \cdot}
\qquad
\frac{\Gamma^\circ \rightsquigarrow \tilde{\Gamma}}{\Gamma^\circ, x :^0 T^0 \rightsquigarrow \tilde{\Gamma}, x :^0 \forall \gamma.[\![T^0]\!]_{\mathsf{rg}(|\tilde{\Gamma}|_1),\gamma}}
$$

$$
\frac{\Gamma^\circ \rightsquigarrow \tilde{\Gamma}}{\Gamma^\circ, x :^1 T^1 \rightsquigarrow \tilde{\Gamma}, x :^1 [\![T^1]\!]}
\qquad
\frac{\Gamma^\circ \rightsquigarrow \tilde{\Gamma}}{\Gamma^\circ \rightsquigarrow \tilde{\Gamma}, \not{x} :^1 \gamma}
$$

Fig. 9. Embedding from $\lambda_{\bigcirc}$

Finally, we prove the soundness of the translation.

Theorem 4 (Soundness of Embedding from $\lambda_{\bigcirc}$).

- *If* $\Gamma^\circ \vdash_0 M^0 \colon T^0$ *and* $\Gamma^\circ \rightsquigarrow \tilde{\Gamma}$, *then* $|\tilde{\Gamma}|_0 \vdash [\![M^0]\!]_{\tilde{\Gamma}} \colon [\![T^0]\!]_{\mathsf{rg}(|\tilde{\Gamma}|_1)}$.
- *If* $\Gamma^\circ \vdash_1 M^1 \colon T^1$ *and* $\Gamma^\circ \rightsquigarrow \tilde{\Gamma}$, *then* $|\tilde{\Gamma}|_0, 🔒, |\tilde{\Gamma}|_1 \vdash [\![M^1]\!]_{\tilde{\Gamma}} \colon [\![T^1]\!]$.

Proof (Sketch). By mutual induction on derivation of $\lambda_{\bigcirc}$.

We focus on the case of level-0 application. If $M^0 = M_1^0\, M_2^0$, then $\Gamma^\circ \vdash_0 M_1^0 \colon S^0 \to T^0$ and $\Gamma^\circ \vdash_0 M_2^0 \colon S^0$ for some S^0. By the induction hypothesis, we have the two $\lambda_{\forall[]}$ judgments below.

- $|\tilde{\Gamma}|_0 \vdash [\![M_1^0]\!]_{\tilde{\Gamma}} \colon (\forall \gamma.[\![S^0]\!]_{\mathsf{rg}(|\tilde{\Gamma}|_1),\gamma}) \to [\![T^0]\!]_{\mathsf{rg}(|\tilde{\Gamma}|_1)}$
- $|\tilde{\Gamma}, \not{x} :^1 \gamma|_0 \vdash [\![M_2^0]\!]_{\tilde{\Gamma}, \not{x} :^1 \gamma} \colon [\![S^0]\!]_{\mathsf{rg}(|\tilde{\Gamma}, \not{x} :^1 \gamma|_1)}$

The second judgment holds because $\Gamma^\circ \rightsquigarrow \tilde{\Gamma}, \not{x} :^1 \gamma$ can be derived from $\Gamma^\circ \rightsquigarrow \tilde{\Gamma}$. We can derive $|\tilde{\Gamma}|_0 \vdash \Lambda \gamma.[\![M_2^0]\!]_{\tilde{\Gamma}, \not{x} :^1 \gamma} \colon \forall \gamma.[\![S^0]\!]_{\mathsf{rg}(|\tilde{\Gamma}, \not{x} :^1 \gamma|_1)}$ from the second judgment considering that $|\tilde{\Gamma}, \not{x} :^1 \gamma|_0 = |\tilde{\Gamma}|_0$. Then we can apply this judgment to the first judgment, and we obtain $|\tilde{\Gamma}|_0 \vdash [\![M_1^0]\!]_{\tilde{\Gamma}}\, (\Lambda \gamma.[\![M_2^0]\!]_{\tilde{\Gamma}, \not{x} :^1 \gamma}) \colon [\![T^0]\!]_{\mathsf{rg}(|\tilde{\Gamma}|_1)}$. ∎

It is worth noting that this embedding requires multiple occurrences of context variables in a single context: As we have seen, $(\bigcirc\mathrm{int} \to \bigcirc\mathrm{str}) \to \bigcirc(\mathrm{int} \to \mathrm{str})$ translates to $(\forall\gamma.(\forall\delta.[\gamma, \delta \vdash \mathrm{int}]) \to [\gamma \vdash \mathrm{str}]) \to [\bullet \vdash \mathrm{int} \to \mathrm{str}]$, where the type $[\gamma, \delta \vdash \mathrm{int}]$ uses two context variables. This fact strongly suggests that context variables in $\lambda_{\forall[]}$ are essential for embedding linear-time temporal types and hence also staged computation.

7 Related Work

Contextual Modal Type Theory. Early work on calculi for metaprogramming with explicit contexts include λ^{poly}_{open} by Kim et al. [12] and $\nu^{\Box}$ by Nanevski and Pfenning [16]. On the one hand, λ^{poly}_{open} has a Fitch-style-like modal type system with explicit contexts and is type safe in the presence of mutable reference and run-time evaluation. On the other hand, $\nu^{\Box}$ has a dual-context-like modal type system that is type sound with run-time evaluation. Both calculi use symbolic representation for named contexts of quoted code. As a result, names in quoted code are not subject to α-conversion. It is worth noting that both papers discuss context polymorphism to achieve flexibility for computation with contexts.

Nanevski et al. refined $\nu^{\Box}$ to contextual modal type theory (CMTT) [17], allowing α-conversion for variables in quoted code. CMTT is very close to our $\lambda_{[]}$ while it employs dual-context style formulation. We believe it is not difficult to apply polymorphic context types to dual-context CMTT, although we do not explore it in this paper. CMTT provides a basis for several metaprogramming languages [9,20,26]. We expect that $\lambda_{\forall[]}$ will contribute to future designs of metaprogramming languages as well.

One notable difference between CMTT and $\lambda_{[]}$ is that CMTT has a *named* context inside a contextual modal type, instead of an (unnamed) context. This approach makes α-conversion somewhat complicated: a CMTT term $box(x\colon T.x)$ has a type $[x\colon T]T$ while an α-equivalent term $box(y\colon T.y)$ has a bit different type $[y\colon T]T$. Instead, $\lambda_{[]}$ omits names from contexts in contextual modal types by identifying variables in a context by their positions; hence α-equivalent terms always have the same type in $\lambda_{[]}$.

Prior Work on Polymorphic Contexts. Contextual modal type systems have been applied to proof assistants [20,3,21,26]. Those proof assistants are designed to allow users to inspect code representation of proof terms using contextual modal types. In particular, Beluga [20,3] allows users to perform pattern match against code with polymorphic contexts, whereas $\lambda_{\forall[]}$ allows only for generative metaprogramming. The prior proposals used an identity substitution id_ϕ as a term representation of a context variable ϕ, whereas we use series variables for that purpose. Type-theoretic formalization of identity substitutions is examined by Puech's unpublished work [23]. He proposed dual-context and Fitch-style contextual modal type theories with polymorphic types and identity substitutions. However, a formalization with identity substitutions introduces a significant restriction: only one occurrence of context variable is allowed in a single context. Suppose we allow multiple occurrences of context variables in a context with identity substitutions. In that case, we have a term like $\mathbf{quo}\langle\gamma,\gamma\rangle(\mathbf{unq}(x)[id_\gamma])$ that is ill-scoped because we do not know which γ is referred to by id_γ. One might consider introducing a restriction that context variable do not duplicate in a context. However, it is still hard to avoid ill-scoped terms like $(\Lambda\delta.\mathbf{quo}\langle\gamma,\delta\rangle(\mathbf{unq}(x)[id_\gamma]))@\gamma$, which reduces to the previous term. That is why we introduce series variables in $\lambda_{\forall[]}$.

Context Subtyping. Rhiger [24] proposed a Fitch-style contextual modal type system $\lambda^{[]}_{<}$ that achieves safe code operation with mutable reference and run-time evaluation. An interesting point of $\lambda^{[]}_{<}$ is that it employs linear-time flavored named contexts where a quote does not discard a future-stage context, and achieves flexibility of computation with context by introducing structural subtyping for contexts. Kiselyov et al. proposed a type system `<NJ>` with a notion of *refined environment classifiers* [14], which can be interpreted as encapsulated representation of contexts. `<NJ>` is similar to $\lambda^{[]}_{<}$ in the sense that it employs classifier subtyping while it is closer to nominal subtyping. They suggested bounded polymorphism over classifiers as potential extension of `<NJ>`, which will allow a type like $\forall\gamma.(\forall\delta \succ \gamma.\langle T_1\rangle^\delta \to \langle T_2\rangle^\delta) \to \langle T_1 \to T_2\rangle^\gamma$. Their bounded polymorphism is likely as expressive as polymorphic contexts of $\lambda_{\forall[]}$, and we are interested in the formal relation between them.

Pattern matching against code Analytic metaprogramming that allows pattern matching against code values is considered beneficial and explored recently [18,28,9]. Especially, Mœbius [9] provides a contextual modal type system capable of pattern matching against open code with polymorphic types. It should be feasible to extend $\lambda_{\forall[]}$ to allow pattern matching against code values, but it is left for future work.

Modal Types for Algebraic Effects and Handlers. ECMTT [32] is an interesting application of contextual modal types to algebraic effects and handlers [22]. It uses contexts to track effects of computations and use explicit substitutions to supply effect handlers. The authors mentioned that ECMTT needs some form of context polymorphism to support effect polymorphism. We expect the polymorphic context types in $\lambda_{\forall[]}$ will provide a basis for such an extension. As our formulation allows multiple occurrences of context variables; hence, we can describe a function that combines computations with different polymorphic effects, e.g., $\forall\gamma,\delta.[\gamma \vdash T] \to [\delta \vdash T] \to [\gamma,\delta \vdash T]$.

Linear-Time Temporal Types. There are several attempts at revealing the relation between contextual modal type theory and linear-time temporal type theory. However, not all of them achieved their goal. For example, Davies [5] pointed out that the translation from λ^{poly}_{open} to $\lambda_{\bigcirc}$, proposed by Kim et al. [12], was not sound for some cases. Puech [23] also claimed a sound translation from λ^{ctx}_{I} to λ^{α} [29], which is an extension of $\lambda_{\bigcirc}$ with environment classifiers, but it did not work for some cases, either. His translation infers hidden contexts by introducing logic variables for unknown contexts and collecting constraints on those logic variables through typing derivations. Consequently, the following judgment fails to translate because f is used in two different scopes, and hence contradicting constraints for f is generated.

$$f :^0 \bigcirc T \to \bigcirc T, g :^0 \bigcirc T \to \bigcirc T \to \bigcirc T, z :^1 T \\ \vdash g\ (\mathbf{quo}((\lambda x : T.\mathbf{unq}(f\mathbf{quo}x))z)(f\mathbf{quo}z))$$

These failing translations conversely indicate that the hypothesis by Davies [5] is right: a sound translation from $\lambda_{\bigcirc}$ requires a full form of context polymorphism as in our $\lambda_{\forall[]}$. Kameyama et al. [10] provided a sound translation from a 2-level fragment of λ^{α} to System F with products and a fixed point operator. Their translation uses polymorphic types to represent unknown contexts, similarly to our approach. However, their translation takes an approach different from ours. For example, a $\lambda_{\bigcirc}$ type $\bigcirc T \to \bigcirc T \to \bigcirc T$ is encoded to $\forall\gamma.([\gamma \vdash T] \to \forall\delta.([\gamma, \delta \vdash T] \to [\gamma, \delta \vdash T]))$ if we apply their approach to $\lambda_{\forall[]}$, whereas the same type is encoded to $(\forall\gamma.[\gamma \vdash T]) \to (\forall\gamma.[\gamma \vdash T]) \to [\bullet \vdash T]$ by the approach discussed in Section 6. There are two major differences between their approach and ours. Firstly, their translation needs to insert *coercion* functions that extend contexts in types in conjunction with polymorphic types. On the contrary, our approach achieves the same goal purely by polymorphic contexts, making the translation much more concise. Secondly, their source language supports richer expressions than $\lambda_{\bigcirc}$, including run-time evaluation and fixpoint. It is left for future work to figure out whether our approach can also embed such features of λ^{α} to $\lambda_{\forall[]}$.

8 Conclusion

This paper has proposed a novel contextual modal type theory $\lambda_{\forall[]}$ with polymorphic contexts. It is novel in that it supports parametric polymorphic contexts and allows us to have multiple context variables in a single context. We have given its semantics by β-reduction and proved subject reduction, strong normalization, and confluence. We have also demonstrated sound embedding from linear-time temporal type theory. We expect that this result shows that $\lambda_{\forall[]}$ endows expressiveness sufficient to describe programs with staged computation.

We regard this work as a first step to establishing a mature modal type theory that reasons hygienic binding operations provided by procedural macros of Scheme, Racket, and several languages. Future work includes formal reasoning of the relation between contextual modal types and refined environment classifiers and developing contextual modal type theory that can express first-class variable names.

Acknowledgements We would like to express gratitude to anonymous referees for their constructive feedback. This work is supported in part by JSPS KAKENHI Grant Number JP20H00582 and JST, the establishment of university fellowships towards the creation of science technology innovation, Grant Number JPMJFS2123.

References

1. Borghuis, V.A.J.: Coming to terms with modal logic: on the interpretation of modalities in typed lambda-calculus. Ph.D. thesis, Technische Universiteit Eindhoven (1994). https://doi.org/10.6100/IR427575
2. Calcagno, C., Taha, W., Huang, L., Leroy, X.: Implementing multi-stage languages using ASTs, gensym, and reflection. In: Pfenning, F., Smaragdakis, Y. (eds.) Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2830, pp. 57–76. Springer (2003). https://doi.org/10.1007/978-3-540-39815-8_4
3. Cave, A., Pientka, B.: First-class substitutions in contextual type theory. In: Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks & Meta-Languages: Theory & Practice. pp. 15–24. LFMTP '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2503887.2503889
4. Clouston, R.: Fitch-style modal lambda calculi. In: Baier, C., Dal Lago, U. (eds.) Proc. of Foundations of Software Science and Computation Structures. pp. 258–275. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-89366-2_14
5. Davies, R.: A temporal logic approach to binding-time analysis. J. ACM **64**(1) (Mar 2017). https://doi.org/10.1145/3011069
6. Davies, R., Pfenning, F.: A modal analysis of staged computation. J. ACM **48**(3), 555–604 (May 2001). https://doi.org/10.1145/382780.382785
7. Ganz, S.E., Sabry, A., Taha, W.: Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In: Pierce, B.C. (ed.) Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001. pp. 74–85. ACM (2001). https://doi.org/10.1145/507635.507646
8. Girard, J.Y., Taylor, P., Lafont, Y.: Proofs and Types. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press (1989)
9. Jang, J., Gélineau, S., Monnier, S., Pientka, B.: Mœbius: Metaprogramming using contextual types: The stage where System F can pattern match on itself. Proc. ACM Program. Lang. **6**(POPL) (Jan 2022). https://doi.org/10.1145/3498700
10. Kameyama, Y., Kiselyov, O., Shan, C.: Closing the stage: From staged code to typed closures. In: PEPM'08 – Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. pp. 147–157 (Dec 2008). https://doi.org/10.1145/1328408.1328430
11. Kavvos, G.A.: Dual-context calculi for modal logic. In: Proc. of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 1–12 (2017). https://doi.org/10.1109/LICS.2017.8005089
12. Kim, I., Yi, K., Calcagno, C.: A polymorphic modal type system for lisp-like multi-staged languages. In: Morrisett, J.G., Jones, S.L.P. (eds.) Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006. pp. 257–268. ACM (2006). https://doi.org/10.1145/1111037.1111060
13. Kiselyov, O.: The design and implementation of BER MetaOCaml: System description. In: Codish, M., Sumii, E. (eds.) Functional and Logic Programming – 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8475, pp. 86–102. Springer (2014). https://doi.org/10.1007/978-3-319-07151-0_6

14. Kiselyov, O., Kameyama, Y., Sudo, Y.: Refined environment classifiers. In: Igarashi, A. (ed.) Proc. of Asian Symposium on Programming Languages and Systems. pp. 271–291. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-47958-3_15
15. Martini, S., Masini, A.: A computational interpretation of modal proofs. In: Proof Theory of Modal Logic, pp. 213–241. Springer Netherlands, Dordrecht (1996). https://doi.org/10.1007/978-94-017-2798-3_12
16. Nanevski, A., Pfenning, F.: Staged computation with names and necessity. J. Funct. Program. **15**(6), 893–939 (Nov 2005). https://doi.org/10.1017/S095679680500568X
17. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Trans. Comput. Logic **9**(3) (Jun 2008). https://doi.org/10.1145/1352582.1352591
18. Parreaux, L., Voizard, A., Shaikhha, A., Koch, C.E.: Unifying analytic and statically-typed quasiquotes. Proc. ACM Program. Lang. **2**(POPL) (Dec 2017). https://doi.org/10.1145/3158101, https://doi.org/10.1145/3158101
19. Pfenning, F., Davies, R.: A judgmental reconstruction of modal logic. Mathematical. Structures in Comp. Sci. **11**(4), 511–540 (Aug 2001). https://doi.org/10.1017/S0960129501003322
20. Pientka, B., Dunfield, J.: Beluga: A framework for programming and reasoning with deductive systems (system description). In: Giesl, J., Hähnle, R. (eds.) Automated Reasoning. pp. 15–21. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_2
21. Pientka, B., Thibodeau, D., Abel, A., Ferreira, F., Zucchini, R.: A type theory for defining logics and proofs. In: Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science. LICS '19, IEEE Press (2019). https://doi.org/10.1109/LICS.2019.8785683
22. Plotkin, G.D., Pretnar, M.: Handlers of algebraic effects. In: Castagna, G. (ed.) Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5502, pp. 80–94. Springer (2009). https://doi.org/10.1007/978-3-642-00590-9_7
23. Puech, M.: A contextual account of staged computations (2016), preprint on webpage at http://cedric.cnam.fr/~puechm/draft_contextual.pdf
24. Rhiger, M.: Staged computation with staged lexical scope. In: Seidl, H. (ed.) Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7211, pp. 559–578. Springer (2012). https://doi.org/10.1007/978-3-642-28869-2_28, https://doi.org/10.1007/978-3-642-28869-2_28
25. Sørensen, M.H., Urzyczyn, P.: The Curry–Howard Isomorphism, Studies in Logic and the Foundations of Mathematics, vol. 149, chap. 4, pp. 77–101. Elsevier (2006). https://doi.org/https://doi.org/10.1016/S0049-237X(06)80005-4, https://www.sciencedirect.com/science/article/pii/S0049237X06800054
26. Stampoulis, A., Shao, Z.: VeriML: Typed computation of logical terms inside a language with effects. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. pp. 333–344. ICFP '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1863543.1863591

27. Stucki, N., Biboudis, A., Odersky, M.: A practical unification of multi-stage programming and macros pp. 14–27 (2018). https://doi.org/10.1145/3278122.3278139, https://doi.org/10.1145/3278122.3278139
28. Stucki, N., Brachthäuser, J.I., Odersky, M.: Multi-stage programming with generative and analytical macros. In: Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. p. 110–122. GPCE 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3486609.3487203, https://doi.org/10.1145/3486609.3487203
29. Taha, W., Nielsen, M.F.: Environment classifiers. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 26–37. POPL '03, Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/604131.604134, https://doi.org/10.1145/604131.604134
30. Tsukada, T., Igarashi, A.: A logical foundation for environment classifiers. Log. Methods Comput. Sci. **6**(4) (2010). https://doi.org/10.2168/LMCS-6(4:8)2010, https://doi.org/10.2168/LMCS-6(4:8)2010
31. Valliappan, N., Ruch, F., Tomé Cortiñas, C.: Normalization for fitch-style modal calculi. Proc. ACM Program. Lang. **6**(ICFP) (Aug 2022). https://doi.org/10.1145/3547649, https://doi.org/10.1145/3547649
32. Zyuzin, N., Nanevski, A.: Contextual modal types for algebraic effects and handlers. Proc. ACM Program. Lang. **5**(ICFP) (Aug 2021). https://doi.org/10.1145/3473580

A Complete Inference System for Skip-free Guarded Kleene Algebra with Tests

Tobias Kappé[1,2](✉), Todd Schmid[3], and Alexandra Silva[4]

[1] Open University of the Netherlands, Heerlen, The Netherlands
tobias.kappe@ou.nl
[2] ILLC, University of Amsterdam, Amsterdam, The Netherlands
[3] University College London, London, UK
[4] Cornell University, Ithaca, NY, USA

Abstract. Guarded Kleene Algebra with Tests (GKAT) is a fragment of Kleene Algebra with Tests (KAT) that was recently introduced to reason efficiently about imperative programs. In contrast to KAT, GKAT does not have an algebraic axiomatization, but relies on an analogue of Salomaa's axiomatization of Kleene Algebra. In this paper, we present an algebraic axiomatization and prove two completeness results for a large fragment of GKAT consisting of *skip-free programs*.

1 Introduction

Kleene algebra with tests (KAT) [26] is a logic for reasoning about semantics and equivalence of simple imperative programs. It extends Kleene Algebra (KA) with Boolean control flow, which enables encoding of conditionals and while loops.

KAT has been applied to verification tasks. For example, it was used in proof-carrying Java programs [24], in compiler optimization [28], and file systems [8]. More recently, KAT was used for reasoning about packet-switched networks, serving as a core to NetKAT [4] and Probabilistic NetKAT [12,43].

The success of KAT in networking is partly due to its dual nature: it can be used to both specify and verify network properties. Moreover, the implementations of NetKAT and ProbNetKAT were surprisingly competitive with state-of-the-art tools [13,44]. Part of the surprise with the efficiency of these implementations is that the decision problem for equivalence in both KAT and NetKAT is PSPACE-complete [29,4]. Further investigations [42] revealed that the tasks performed in NetKAT only make use of a fragment of KAT. It turns out that the difficulty of deciding equivalence in KAT can largely be attributed to the non-deterministic nature of KAT programs. If one restricts to KAT programs that operate deterministically with respect to Boolean control flow, the associated decision problem is almost linear. This fragment of KAT was first identified in [30] and further explored as *guarded Kleene algebra with tests* (GKAT) [42].

The study in [42] proved that the decision problem for GKAT programs is almost linear, and proposed an axiomatization of equivalence. However, the axiomatization suffered from a serious drawback: it included a powerful *uniqueness*

The original version of this chapter was revised: The order of the author names has been corrected to alphabetical order. The correction to this chapter is available at https://doi.org/10.1007/978-3-031-30044-8_21

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 309–336, 2023.
https://doi.org/10.1007/978-3-031-30044-8_12

of solutions axiom (UA), which greatly encumbers algebraic reasoning in practice. In order to use (UA) to show that a pair of programs are equivalent, one needs to find a system of equations that they both satisfy. Even more worryingly, the axiomatization contained a fixed-point axiom with a side condition reminiscent of Salomaa's axiomatization for regular expressions, which is known to be non-algebraic and impair the use of the axiomatic reasoning in context (as substitution of atomic programs is not sound anymore). The authors of [42] left as open questions whether (UA) can be derived from the other GKAT axioms and whether the non-algebraic side condition can be removed. Despite the attention GKAT has received in recent literature [40,48,41], these questions remain open.

In the present work, we offer a partial answer to the questions posed in [42]. We show that proving the validity of an equivalence in GKAT does not require (UA) if the pair of programs in question are of a particular form, what we call *skip-free*. This fragment of GKAT is expressive enough to capture a large class of programs, and it also provides a better basis for algebraic reasoning: we show that the side condition of the fixed-point axiom can be removed. Our inspiration to look at this fragment came from recent work of Grabmayer and Fokkink's on the axiomatization of *1-free star expressions modulo bisimulation* [15,14], an important stepping stone to solving a decades-open problem posed by Milner [33].

In a nutshell, our contribution is to identify a large fragment of GKAT, what we call the *skip-free fragment*, that admits an algebraic axiomatization. We axiomatize both bisimilarity and language semantics and provide two completeness proofs. The first proves completeness of skip-free GKAT *modulo bisimulation* [40], via a reduction to completeness of Grabmayer and Fokkink's system [15]. The second proves completeness of skip-free GKAT w.r.t. language semantics via a reduction to skip-free GKAT modulo bisimulation. We also show that equivalence proofs of skip-free GKAT expressions (for both semantics) embed in full GKAT.

The next section contains an introduction to GKAT and an overview of the open problems we tackle in the technical sections of the paper.

2 Overview

In this section we provide an overview of our results. We start with a motivating example of two imperative programs to discuss program equivalence as a verification technology. We then show how GKAT can be used to solve this problem and explore the open questions that we tackle in this paper.

Equivalence for Verification. In the game *Fizz! Buzz!* [36], players sit in a circle taking turns counting up from one. Instead of saying any number that is a multiple of 3, players must say "fizz", and multiples of 5 are replaced with "buzz". If the number is a multiple both 3 and 5, the player must say "fizz buzz".

Imagine you are asked in a job interview to write a program that prints out the first 100 rounds of a perfect game of *Fizz! Buzz!*. You write the function fizzbuzz1 as given in Figure 1(i). Thinking about the interview later that day, you look up a solution, and you find fizzbuzz2, depicted in Figure 1(ii). You

```
def fizzbuzz1 =                                    (i)
  n := 1;
  while n ≤ 100 do
    if 3|n then
      if not 5|n then
        print fizz; n++;
      else
        print fizzbuzz; n++;
    else if 5|n then
      print buzz; n++;
    else
      print n; n++;
  print done!;
```

```
def fizzbuzz2 =                                    (ii)
  n := 1;
  while n ≤ 100 do
    if 5|n and 3|n then
      print fizzbuzz;
    else if 3|n then
      print fizz;
    else if 5|n then
      print buzz;
    else
      print n;
    n++;
  print done!;
```

Fig. 1. Two possible specifications of the ideal *Fizz! Buzz!* player.

suspect that fizzbuzz2 should do the same thing as fizzbuzz1, and after thinking it over for a few minutes, you realize your program could be transformed into the reference solution by a series of transformations that do not change its semantics:

1. Place the common action n++ at the end of the loop.
2. Replace not $5|n$ with $5|n$ and swap print *fizz* with print *fizzbuzz*.
3. Merge the nested branches of $3|n$ and $5|n$ into one.

Feeling somewhat more reassured, you ponder the three steps above. It seems like their validity is independent of the actual tests and actions performed by the code; for example, swapping the branches of an if - then - else - block while negating the test should be valid under *any* circumstances. This raises the question: is there a family of primitive transformations that can be used to derive valid ways of rearranging imperative programs? Furthermore, is there an algorithm to decide whether two programs are equivalent under these laws?

Enter GKAT. Guarded Kleene Algebra with Tests (GKAT) [42] has been proposed as a way of answering the questions above. Expressions in the language of GKAT model skeletons of imperative programs, where the exact meaning of tests and actions is abstracted. The laws of GKAT correspond to program transformations that are valid regardless of the semantics of tests and actions.

Formally, GKAT expressions are captured by a two-level grammar, generated by a finite set of tests T and a finite set of actions Σ, as follows:

$$\mathsf{BExp} \ni b, c ::= 0 \mid 1 \mid t \in T \mid b \vee c \mid b \wedge c \mid \bar{b}$$
$$\mathsf{GExp} \ni e, f ::= p \in \Sigma \mid b \mid e +_b f \mid e \cdot f \mid e^{(b)}$$

BExp is the set of *Boolean expressions*, built from 0 (false), 1 (true), and primitive tests from T, and composed using $\vee$ (or), $\wedge$ (and) and $\bar{}$ (not). GExp is the set of GKAT *expressions*, built from tests (assert statements) and primitive actions $p \in \Sigma$. Here, $e +_b f$ is a condensed way of writing 'if b then e else f', and $e^{(b)}$ is shorthand for 'while b do e'; the operator $\cdot$ models sequential composition. By convention, the sequence operator $\cdot$ takes precedence over the operator $+_b$.

Example 2.1. Abbreviating statements of the form $\mathsf{print}\ \mathit{foo}$ by simply writing foo, Figure 1(i) can be rendered as the GKAT expression

$$(n := 1) \cdot \begin{pmatrix} (\mathit{fizz} \cdot n{+}{+} +_{\overline{5|n}} \mathit{fizzbuzz} \cdot n{+}{+}) +_{3|n} \\ (\mathit{buzz} \cdot n{+}{+} +_{5|n} n \cdot n{+}{+}) \end{pmatrix}^{(n \leq 100)} \cdot \mathit{done!} \tag{1}$$

Similarly, the program in Figure 1(ii) gives the GKAT expression

$$(n := 1) \cdot ((\mathit{fizzbuzz} +_{5|n \wedge 3|n} (\mathit{fizz} +_{3|n} (\mathit{buzz} +_{5|n} n))) \cdot n{+}{+})^{(n \leq 100)} \cdot \mathit{done!} \tag{2}$$

Semantics. A moment ago, we stated that GKAT equivalences are intended to witness program equivalence, regardless of how primitive tests and actions are interpreted. We make this more precise by recalling the *relational* semantics of GKAT programs [42].[5] The intuition behind this semantics is that if the possible states of the machine being programmed are modelled by some set S, then tests are predicates on S (comprised of all states where the test succeeds), and actions are relations on S (encoding the changes in state affected by the action).

Definition 2.2 ([42]). *A* (relational) interpretation *is a triple* $\sigma = (S, \mathsf{eval}, \mathsf{sat})$ *where S is a set,* $\mathsf{eval} : \Sigma \to \mathcal{P}(S \times S)$ *and* $\mathsf{sat} : T \to \mathcal{P}(S)$*. Each relational interpretation σ gives rise to a semantics* $[\![-]\!]_\sigma : \mathsf{GExp} \to \mathcal{P}(S \times S)$*, as follows:*

$$\begin{aligned} [\![0]\!]_\sigma &= \emptyset & [\![\bar{a}]\!]_\sigma &= [\![1]\!]_\sigma \setminus [\![a]\!]_\sigma \\ [\![1]\!]_\sigma &= \{(s,s) : s \in S\} & [\![p]\!]_\sigma &= \mathsf{eval}(p) \\ [\![t]\!]_\sigma &= \{(s,s) : s \in \mathsf{sat}(t)\} & [\![e +_b f]\!]_\sigma &= [\![b]\!]_\sigma \circ [\![e]\!]_\sigma \cup [\![\bar{b}]\!]_\sigma \circ [\![f]\!]_\sigma \\ [\![b \wedge c]\!]_\sigma &= [\![b]\!]_\sigma \cap [\![c]\!]_\sigma & [\![e \cdot f]\!]_\sigma &= [\![e]\!]_\sigma \circ [\![f]\!]_\sigma \\ [\![b \vee c]\!]_\sigma &= [\![b]\!]_\sigma \cup [\![c]\!]_\sigma & [\![e^{(b)}]\!]_\sigma &= ([\![b]\!]_\sigma \circ [\![e]\!]_\sigma)^* \circ [\![\bar{b}]\!]_\sigma \end{aligned}$$

Here we use $\circ$ for relation composition and * for reflexive transitive closure.

Remark 2.3. If $\mathsf{eval}(p)$ is a partial function for every $p \in \Sigma$, then so is $[\![e]\!]_\sigma$ for each e. The above therefore also yields a semantics in terms of partial functions.

The relation $[\![e]\!]_\sigma$ contains the possible pairs of start and end states of the program e. For instance, the input-output relation of $[\![e +_b f]\!]$ consists of the pairs in $[\![e]\!]_\sigma$ (resp. $[\![f]\!]_\sigma$) where the start state satisfies b (resp. violates b).

Example 2.4. We could model the states of the machine running *Fizz! Buzz!* as pairs (m, ℓ), where m is the current value of the counter n, and ℓ is a list of words printed so far; the accompanying maps sat and eval are given by:

$$\begin{aligned} \mathsf{sat}(k|n) &= \{(m,\ell) \in S : m \equiv 0 \bmod k\} \\ \mathsf{sat}(n \leq k) &= \{(m,\ell) \in S : m \leq k\} \\ \mathsf{eval}(n{+}{+}) &= \{((m,\ell),(m+1,\ell) : (m,\ell) \in S\} \\ \mathsf{eval}(n := k) &= \{((m,\ell),(k,\ell)) : (m,\ell) \in S)\} \\ \mathsf{eval}(\mathsf{w}) &= \{((m,\ell),(m,\ell\mathsf{w})) : (m,\ell) \in S\} \qquad (\mathsf{w} \in \{\mathit{fizz}, \mathit{buzz}, \mathit{fizzbuzz}\}) \\ \mathsf{eval}(n) &= \{((m,\ell),(m,\ell m)) : (m,\ell) \in S\} \end{aligned}$$

[5] A probabilistic semantics in terms of sub-Markov kernels is also possible [42].

For instance, the interpretation of n++ connects states of the form (m, ℓ) to states of the form $(m+1, \ell)$—incrementing the counter by one, and leaving the output unchanged. Similarly, print statements append the given string to the output.

On the one hand, this parameterized semantics shows that programs in the GKAT syntax can be given a semantics that corresponds to the intended meaning of their actions and tests. On the other hand, it allows us to quantify over all possible interpretations, and thus abstract from the meaning of the primitives.

As it happens, two expressions have the same relational semantics under any interpretation if and only if they have the same *language semantics* [42], i.e., in terms of languages of *guarded strings* as used in KAT [26]. Since equivalence under the language semantics is efficiently decidable [42], so is equivalence under all relational interpretations. The decision procedure in [42] uses bisimulation and known results from automata theory. These techniques are good for mechanization but hide the algebraic structure of programs that plays. To expose this, algebraic laws of GKAT program equivalence were studied.

Program transformations. GKAT programs are (generalized) regular expressions, which are intuitive to reason about and for which many syntactic equivalences are known and explored. In [42], a set of sound axioms $e \equiv f$ such that $[\![e]\!]_\sigma = [\![f]\!]_\sigma$ for all σ was proposed, and it was shown that these can be used to prove a number of useful facts about programs. For instance, the following two equivalences are axioms of GKAT:

$$e \cdot g +_b f \cdot g \equiv (e +_b f) \cdot g \qquad\qquad f +_{\overline{b}} e \equiv e +_b f$$

The first of these says that common code at the tail end of branches can be factored out, while the second says that the code in branches of a conditional can be swapped, as long as we negate the test. Returning to our running example, if we apply the first law to (1) three times (once for each guarded choice),

$$(n := 1) \cdot \left(\left(\begin{matrix} (\mathit{fizzbuzz} +_{5|n} \mathit{fizz}) +_{3|n} \\ (\mathit{buzz} +_{5|n} n) \end{matrix} \right) \cdot n\text{++} \right)^{(n \leq 100)} \cdot \mathit{done!} \tag{3}$$

Finally, we can apply $(e +_b f) +_c (g +_b h) \equiv e +_{b \wedge c} (f +_c (g +_b h))$, which is provable from the axioms of GKAT, to transform (3) into (2).

Being able to transform one GKAT program into another using the axioms of GKAT is useful, but the question arises: do the axioms capture *all* equivalences that hold? More specifically, are the axioms of GKAT powerful enough to prove that $e \equiv f$ whenever $[\![e]\!]_\sigma = [\![f]\!]_\sigma$ holds for all σ?

In [42], a partial answer to the above question is provided: if we extend the laws of GKAT with the *uniqueness axiom* (UA), then the resulting set of axioms is sound and complete w.r.t. the language semantics. The problem with this is that (UA) is not really a single axiom, but rather an *axiom scheme*, which makes both its presentation and application somewhat unwieldy.

To properly introduce (UA), we need the following notion.

Definition 2.5. *A* left-affine system *is defined by expressions* $e_{11}, \ldots, e_{nn} \in \mathsf{GExp}$ *and* $f_1, \ldots, f_n \in \mathsf{GExp}$*, along with tests* $b_{11}, \ldots, b_{nn} \in \mathsf{BExp}$*. A sequence of expressions* $s_1, \ldots, s_n \in \mathsf{GExp}$ *is said to be a* solution *to this system if*

$$s_i \equiv e_{i1} \cdot s_1 +_{b_{i1}} e_{i2} \cdot s_2 +_{b_{i2}} \cdots +_{b_{i(n-1)}} e_{in} +_{b_{in}} f_i \quad (\forall i \leq n)$$

Here, the operations $+_{b_{ij}}$ *associate to the right.*

A left-affine system is called guarded *if no* e_{ij} *that appears in the system successfully terminates after reading an atomic test. In other words, each coefficient denotes a productive program, meaning it must execute some action before successfully terminating—we refer to Section 7.3 for more details.*

Stated fully, (UA) says that if expressions $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ are solutions to the same guarded left-affine system, then $s_i \equiv t_i$ for $1 \leq i \leq n$.

On top of the infinitary nature of (UA), the side condition demanding guardedness prevents purely algebraic reasoning: replacing action symbols in a valid GKAT equation with arbitrary GKAT expressions might yield an invalid equation! The situation is analogous to the *empty word property* used by Salomaa [38] to axiomatize equivalence of regular expressions. The side condition of guardedness appearing in (UA) is inherited from another axiom of GKAT, the fixed-point axiom, which in essence is the unary version of this axiom scheme and explicitly defines the solution of one guarded left-affine equation as a while loop.

$$g \equiv eg +_b f \Longrightarrow g \equiv e^{(b)} f \qquad \text{if } e \text{ is guarded.}$$

Remark 2.6. Part of the problem of the uniqueness axiom is that the case for general n does not seem to follow easily from the case where $n = 1$. The problem here is that, unlike the analogous situation for Kleene algebra, there is no general method to transform a left-affine system with $n + 1$ unknowns into one with n unknowns [30], even if this is possible in certain cases [42].

The open questions. We are motivated by two open questions from [42]:

– First, can the uniqueness axiom be eliminated? The other axioms of GKAT contain the instantiation of (UA) for $n = 1$, which has so far been sufficient in all handwritten proofs of equivalence that we know. Yet (UA) seems to be necessary in both known completeness proofs.
– Second, can we eliminate the guardedness side condition? Kozen [25] showed that Salomaa's axiomatization is subsumed by a set of axioms that together imply existence and uniqueness of *least* solutions to systems of equations, but this approach has not yet borne fruit in GKAT.

This paper. Our main contribution is to show that, in a particular fragment of GKAT, both questions can be answered in the positive (see Figure 2).

In Section 3, we present what we call the skip-free fragment of GKAT, consisting of programs that do not contain assert statements in the body (other than assert false); in other words, Boolean statements are restricted to control statements. For this fragment, we show that the axiom scheme (UA) can be avoided

Guarded Union	**Sequencing**	**Loops**
$x = x +_b x$	$0x = 0$	$x^{(b)}y = x(x^{(b)}y) +_b y$
$x = x +_1 y$	$x0 \overset{(\dagger)}{=} 0$	
$x +_b y = y +_{\bar{b}} x$	$x(yz) = (xy)z$	$\dfrac{z = xz +_b y}{z = x^{(b)}y}$
$x +_b (y +_c z) = (x +_b y) +_{b \vee c} z$	$(x +_b y)z = xz +_b yz$	

Fig. 2. Axioms for language semantics skip-free GKAT (in addition to Boolean algebra axioms for tests, see Fig. 3). If the axiom marked † is omitted the above axiomatize a finer semantics, bisimilarity.

entirely. In fact, this is true for language semantics (as first introduced in [42]) as well as for the bisimulation semantics of [40].

In Section 4, we provide a bridge to a recent result in process algebra. In the 80s, Milner offered an alternative interpretation of regular expressions [33], as what he called *star behaviours*. Based on work of Salomaa from the 1960s [38], Milner proposed a sound axiomatization of the *algebra of star behaviours*, but left completeness an open problem. After 38 years, it was recently solved by Clemens Grabmayer [14] following up on his joint work with Wan Fokkink showing that a suitable restriction of Milner's axioms is complete for the *one-free fragment* of regular expressions modulo bisimulation [15]. We leverage their work with an interesting embedding of skip-free GKAT into the one-free regular expressions.

This leads to two completeness results. In Section 5, we start by focusing on the *bisimulation semantics* of the skip-free fragment, and then in Section 6 expand our argument to its *language semantics*. More precisely, we first provide a reduction of the completeness of skip-free GKAT *up to bisimulation* to the completeness of Grabmayer and Fokkink's 1-free regular expressions modulo bisimulation [15]. We then provide a reduction of the completeness of skip-free GKAT modulo language semantics to the completeness of skip-free GKAT modulo bisimulation via a technique inspired by the *tree pruning* approach of [40].

Finally, in Section 7, we connect our semantics of skip-free GKAT expressions to the established semantics of full GKAT. We also connect the syntactic proofs between skip-free GKAT expressions in both our axiomatization and the existing one. In conjunction with the results of Sections 5 and 6, the results in Section 7 make a significant step towards answering the question of whether the axioms of GKAT give a complete description of program equivalence, in the positive.

Proofs appear in the full version [22].

3 Introducing Skip-free GKAT

The axiom scheme (UA) can be avoided entirely in a certain fragment of GKAT, both for determining bisimilarity and language equivalence. In this section, we give a formal description of the expressions in this fragment and their semantics.

Skip-free expressions. The fragment of GKAT in focus is the one that *excludes sub-programs that may accept immediately*, without performing any action. Since these programs can be "skipped" under certain conditions, we call the fragment

$$\begin{array}{llll} x \vee 0 = x & x \vee \bar{x} = 1 & x \vee y = y \vee x & x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z) \\ x \wedge 1 = x & x \wedge \bar{x} = 0 & x \wedge y = y \wedge x & x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \end{array}$$

Fig. 3. The axioms of Boolean algebra [18].

that avoids them *skip-free*. Among others, it prohibits sub-programs of the form assert b for b $\neq$ false, but also while false do p, which is equivalent to assert true.

Definition 3.1. *Given a set Σ of* atomic actions, *the set* GExp^- *of* skip-free GKAT expressions *is given by the grammar*

$$\mathsf{GExp}^- \ni e_1, e_2 ::= 0 \mid p \in \Sigma \mid e_1 +_b e_2 \mid e_1 \cdot e_2 \mid e_1^{(b)} e_2$$

where b ranges over the Boolean algebra expressions BExp*.*

Unlike full GKAT, in skip-free GKAT the loop construct is treated as a binary operation, analogous to Kleene's original star operation [23], which was also binary. This helps us avoid loops of the form $e^{(b)}$, which can be skipped when b does not hold. The expression $e_1^{(b)} e_2$ corresponds to $e_1^{(b)} \cdot e_2$ in GKAT.

Example 3.2. Using the same notational shorthand as in Example 2.1, the block of code in Figure 1(ii) can be cast as the skip-free GKAT expression

$$(n := 1) \cdot ((\mathit{fizzbuzz} +_{3|n \wedge 5|n} (\mathit{fizz} +_{3|n} (\mathit{buzz} +_{5|n} n))) \cdot n{+}{+})^{(n \leq 100)} (\mathit{done!})$$

Note how we use a skip-free loop of the form ${e_1}^{(b)} e_2$ instead of the looping construct $e_1^{(b)}$ before concatenating with e_2, as was done for GKAT.

3.1 Skip-free Semantics

There are three natural ways to interpret skip-free GKAT expressions: as *automata*, as *behaviours*, and as *languages*.[6] After a short note on Boolean algebra, we shall begin with the automaton interpretation, also known as the *small-step semantics*, from which the other two can be derived.

Boolean algebra. To properly present our automata, we need to introduce one more notion. Boolean expressions BExp are a syntax for elements of a *Boolean algebra*, an algebraic structure satisfying the equations in Fig. 3. When a Boolean algebra is freely generated from a finite set of basic tests (T in the case of BExp), it has a finite set At of nonzero minimal elements called *atoms*. Atoms are in one-to-one correspondence with sets of tests, and the Boolean algebra is isomorphic to $\mathcal{P}(\mathsf{At})$, the sets of subsets of At, equipped with $\vee = \cup$, $\wedge = \cap$, and $\overline{(-)} = \mathsf{At} \setminus (-)$. In the context of programming, one can think of an atom as a complete description of the machine state, saying which tests are true and which are false. We will denote atoms by the Greek letters α and β, sometimes with indices. Given a Boolean expression $b \in \mathsf{BExp}$ and an atom $\alpha \in \mathsf{At}$ we say that α entails b, written $\alpha \leq b$, whenever $\overline{\alpha} \vee b = 1$, or equivalently $\alpha \vee b = b$.

[6] We will connect these to the relational semantics from Definition 2.2 in Section 7.

$$\frac{}{p \xrightarrow{\alpha|p} \checkmark} \qquad \frac{e_1 \xrightarrow{\alpha|p} e' \quad \alpha \leq b}{e_1 +_b e_2 \xrightarrow{\alpha|p} e'} \qquad \frac{e_2 \xrightarrow{\alpha|p} e' \quad \alpha \not\leq b}{e_1 +_b e_2 \xrightarrow{\alpha|p} e'} \qquad \frac{e_1 \xrightarrow{\alpha|p} e'}{e_1 e_2 \xrightarrow{\alpha|p} e' e_2}$$

$$\frac{e_1 \xrightarrow{\alpha|p} \checkmark}{e_1 e_2 \xrightarrow{\alpha|p} e_2} \qquad \frac{e_1 \xrightarrow{\alpha|p} e' \quad \alpha \leq b}{e_1^{(b)} e_2 \xrightarrow{\alpha|p} e'(e_1^{(b)} e_2)} \qquad \frac{e_1 \xrightarrow{\alpha|p} \checkmark \quad \alpha \leq b}{e_1^{(b)} e_2 \xrightarrow{\alpha|p} e_1^{(b)} e_2} \qquad \frac{e_2 \xrightarrow{\alpha|p} e' \quad \alpha \not\leq b}{e_1^{(b)} e_2 \xrightarrow{\alpha|p} e'}$$

Fig. 4. The small-step semantics of skip-free GKAT expressions.

Automata. Throughout the paper, we use the notation $\bullet + S$ where S is a set and $\bullet$ is a symbol to denote the disjoint union (coproduct) of $\{\bullet\}$ and S.

The small-step semantics of a skip-free GKAT expression uses a special type of deterministic automaton. A *skip-free automaton* is a pair (X, h), where X is a set of *states* and $h \colon X \to (\bot + \Sigma \times (\checkmark + X))^{\mathsf{At}}$ is a *transition structure*. At every $x \in X$ and for any $\alpha \in \mathsf{At}$, one of three things can happen:

1. $h(x)(\alpha) = (p, y)$, which we write as $x \xrightarrow{\alpha|p} y$, means the state x under α makes a transition to a new state y, after performing the action p;
2. $h(x)(\alpha) = (p, \checkmark)$, which we write $x \xrightarrow{\alpha|p} \checkmark$, means the state x under α successfully terminates with action p;
3. $h(x)(\alpha) = \bot$, which we write $x \downarrow \alpha$, means the state x under α terminates with failure. Often we will leave these outputs implicit.

Definition 3.3 (Automaton of expressions). *We equip the set* GExp^- *of all skip-free GKAT expressions with an automaton structure* $(\mathsf{GExp}^-, \partial)$ *given in Fig. 4, representing step-by-step execution. Given* $e \in \mathsf{GExp}^-$*, we denote the set of states reachable from* e *by* $\langle e \rangle$ *and call this the* small-step semantics *of* e*.*

The small-step semantics of skip-free GKAT expressions is inspired by Brzozowski's *derivatives* [7], which provide an automata-theoretic description of the step-by-step execution of a regular expression. Our first lemma tells us that, like regular expressions, skip-free GKAT expressions correspond to finite automata.

Lemma 3.4. *For any* $e \in \mathsf{GExp}^-$*,* $\langle e \rangle$ *has finitely many states.*

Example 3.5. The automaton that arises from the program fizzbuzz2 is below, with $a = n \leq 100$, $b = 3|n$, and $c = 5|n$. The expression e is the same as in Example 3.2, e_1 is the same as e but without the action $n := 0$ in front, and $e_2 = n{+}{+} \cdot e_1$. We also adopt the convention of writing $x \xrightarrow{b|p} x'$ where $b \in \mathsf{BExp}$ to represent all transitions $x \xrightarrow{\alpha|p} x'$ where $\alpha \leq b$.

$$\begin{tikzcd}
 & \checkmark & \\
e \arrow[r, "1 \mid n := 1"'] & e_1 \arrow[u, "\bar{a} \mid \textit{done!}"] \arrow[r, bend left, "{abc \mid \textit{fizzbuzz},\ ab\bar{c} \mid \textit{fizz},\ a\bar{b}c \mid \textit{buzz},\ a\bar{b}\bar{c} \mid n}"] & e_2 \arrow[l, "1 \mid n{+}{+}"']
\end{tikzcd}$$

The automaton interpretation of a skip-free GKAT expression (its small-step semantics) provides an intuitive visual depiction of the details of its execution. This is a useful view on the operational semantics of expressions, but sometimes one might want to have a more precise description of the global behaviour of the program. The remaining two interpretations of skip-free GKAT expressions aim to capture two denotational semantics of expressions: one finer, bisimilarity, that makes a distinction on the branching created by how its states respond to atomic tests, which actions can be performed, and when successful termination and crashes occur; another coarser, language semantics, that assigns a language of traces to each expression capturing all sequences of actions that lead to successful termination. The key difference between these two semantics is their ability to distinguish programs that crash early in the execution versus programs that crash later—this is evident in the axiomatizations of both semantics. We start by presenting the language semantics as this is the more traditional one associated with GKAT (and regular) expressions.

Language semantics. Formally, a *(skip-free) guarded trace* is a nonempty string of the form $\alpha_1 p_1 \cdots \alpha_n p_n$, where each $\alpha_i \in \mathsf{At}$ and $p_i \in \Sigma$. Intuitively, each α_i captures the state of program variables needed to execute program action p_i and the execution of each p_i except the last yields a new program state α_{i+1}. A *skip-free guarded language* is a set of guarded traces.

Skip-free guarded languages should be thought of as sets of strings denoting successfully terminating computations.

Definition 3.6 (Language acceptance). *In a skip-free automaton (X, h) with a state $x \in X$, the* language accepted by *x is the skip-free guarded language*

$$\mathcal{L}(x, (X, h)) = \{\alpha_1 p_1 \cdots \alpha_n p_n \mid x \xrightarrow{\alpha_1 | p_1} x_1 \to \cdots \to x_n \xrightarrow{\alpha_n | p_n} \checkmark\}$$

If (X, h) is clear from context, we will simply write $\mathcal{L}(x)$ instead of $\mathcal{L}(x, (X, h))$. If $\mathcal{L}(x) = \mathcal{L}(y)$, we write $x \sim_{\mathcal{L}} y$ and say that x and y are language equivalent.

Each skip-free GKAT expression is a state in the automaton of expressions (Definition 3.3) and therefore accepts a language. The language accepted by a skip-free GKAT expression is the set of successful runs of the program it denotes. Analogously to GKAT, we can describe this language inductively.

Lemma 3.7. *Given an expression $e \in \mathsf{GExp}^-$, the language accepted by e in $(\mathsf{GExp}^-, \partial)$, i.e., $\mathcal{L}(e) = \mathcal{L}(e, (\mathsf{GExp}^-, \partial))$ can be characterized as follows:*

$$\mathcal{L}(0) = \emptyset \quad \mathcal{L}(p) = \{\alpha p \mid \alpha \in \mathsf{At}\} \quad \mathcal{L}(e_1 +_b e_2) = b\mathcal{L}(e_1) \cup \bar{b}\mathcal{L}(e_2)$$
$$\mathcal{L}(e_1 \cdot e_2) = \mathcal{L}(e_1) \cdot \mathcal{L}(e_2) \quad \mathcal{L}(e_1^{(b)} e_2) = \bigcup_{n \in \mathbb{N}} (b\mathcal{L}(e_1))^n \cdot \bar{b}\mathcal{L}(e_2)$$

Here, we write $bL = \{\alpha p w \in L \mid \alpha \leq b\}$ and $L_1 \cdot L_2 = \{wx : w \in L_1, x \in L_2\}$, while $L^0 = \{\epsilon\}$ (where ϵ denotes the empty word) and $L^{n+1} = L \cdot L^n$.

Lemma 3.7 provides a way of computing the language of an expression e without having to generate the automaton for e.

Bisimulation semantics. Another, finer, notion of equivalence that we can associate with skip-free automata is bisimilarity.

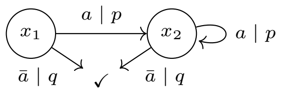
Definition 3.8. *Given skip-free automata* (X, h) *and* (Y, k)*, a* bisimulation *is a relation* $R \subseteq X \times Y$ *such that for any* $x \mathrel{R} y$*,* $\alpha \in \mathsf{At}$ *and* $p \in \Sigma$*:*

1. $x \downarrow \alpha$ *if and only if* $y \downarrow \alpha$*,*
2. $x \xrightarrow{\alpha|p} \checkmark$ *if and only if* $y \xrightarrow{\alpha|p} \checkmark$*, and*
3. *for any* $x' \mathrel{R} y'$*,* $x \xrightarrow{\alpha|p} x'$ *if and only if* $y \xrightarrow{\alpha|p} y'$*.*

We call x *and* y bisimilar *if* $x \mathrel{R} y$ *for some bisimulation* R *and write* $x \leftrightarroweq y$*.*

In a fixed skip-free automaton (X, h), we define $\leftrightarroweq \subseteq X \times X$ to be the largest bisimulation, called *bisimilarity.* This is an equivalence relation and a bisimulation.[7] The bisimilarity equivalence class of a state is often called its *behaviour.*

Example 3.9. In the automaton below, x_1 and x_2 are bisimilar. This is witnessed by the bisimulation $\{(x_1, x_2), (x_2, x_2)\}$.



We can also use bisimulations to witness language equivalence.

Lemma 3.10. *Let* $e_1, e_2 \in \mathsf{GExp}^-$*. If* $e_1 \leftrightarroweq e_2$*, then* $\mathcal{L}(e_1) = \mathcal{L}(e_2)$*.*

The converse of Lemma 3.10 is not true. Consider, for example, the program $p^{(1)}q$ that repeats the atomic action $p \in \Sigma$ indefinitely, never reaching q. Since

$$\mathcal{L}(p^{(1)}q) = \bigcup_{n\in\mathbb{N}} \mathcal{L}(p)^n \cdot \emptyset = \emptyset = \mathcal{L}(0)$$

we know that $p^{(1)}q \sim_{\mathcal{L}} 0$. But $p^{(1)}q$ and 0 are not bisimilar, since Fig. 4 tells us that $p^{(1)}q \xrightarrow{\alpha|p} p^{(1)}q$ and $0 \downarrow \alpha$, which together refute Definition 3.8.1.

3.2 Axioms

Next, we give an inference system for bisimilarity and language equivalence consisting of equations and equational inference rules. The axioms of skip-free GKAT are given in Fig. 2. They include the equation (†), which says that early deadlock is the same as late deadlock. This is *sound* with respect to the language interpretation, meaning that (†) is true if x is replaced with a skip-free guarded language, but it is not sound with respect to the bisimulation semantics. For example, the expressions $p \cdot 0$ and 0 are not bisimilar for any $p \in \Sigma$. Interestingly, this is the only axiomatic difference between bisimilarity and language equivalence.

[7] This follows directly from seeing skip-free automata as a special type of coalgebra and the fact that the functor involved preserves weak pullbacks [37]. In fact, coalgebra has been an indispensable tool in the production of the current paper, guiding us to the correct definitions and simplifying many of the proofs.

Remark 3.11. The underlying logical structure of our inference systems is equational logic [5], meaning that provable equivalence is an equivalence relation that is preserved by the algebraic operations.

Given expressions $e_1, e_2 \in \mathsf{GExp}^-$, we write $e_1 \equiv_\dagger e_2$ and say that e_1 and e_2 are $\equiv_\dagger$-*equivalent* if the equation $e_1 = e_2$ can be derived from the axioms in Fig. 2 without the axiom marked (†). We write $e_1 \equiv e_2$ and say that e_1 and e_2 are $\equiv$-*equivalent* if $e_1 = e_2$ can be derived from the whole set of axioms in Fig. 2.

The axioms in Fig. 2 are sound with respect to the respective semantics they axiomatize. The only axiom that is not sound w.r.t. bisimilarity is $x \cdot 0 \equiv 0$, as this would relate automata with different behaviours (x may permit some action to be performed, and this is observable in the bisimulation).

Theorem 3.12 (Soundness). *For any $e_1, e_2 \in \mathsf{GExp}^-$,*

1. *If $e_1 \equiv_\dagger e_2$, then $e_1 \underline{\leftrightarrow} e_2$.*
2. *If $e_1 \equiv e_2$, then $e_1 \sim_{\mathcal{L}} e_2$.*

We consider the next two results, which are jointly converse to Theorem 3.12, to be the main theorems of this paper. They state that the axioms in Fig. 2 are *complete* for bisimilarity and language equivalence respectively, i.e., they describe a complete set of program transformations for skip-free GKAT.

Theorem 3.13 (Completeness I). *If $e_1 \underline{\leftrightarrow} e_2$, then $e_1 \equiv_\dagger e_2$.*

Theorem 3.14 (Completeness II). *If $e_1 \sim_{\mathcal{L}} e_2$, then $e_1 \equiv e_2$.*

We prove Theorem 3.13 in Section 5 by drawing a formal analogy between skip-free GKAT and a recent study of regular expressions in the context of process algebra [15]. We include a short overview of this recent work in the next section.

We delay the proof of Theorem 3.14 to Section 6, which uses a separate technique based on the pruning method introduced in [40].

4 1-free Star Expressions

Regular expressions were introduced by Kleene [23] as a syntax for the algebra of regular events. Milner offered an alternative interpretation of regular expressions [33], as what he called *star behaviours*. Based on work of Salomaa [38], Milner proposed a sound axiomatization of the algebra of star behaviours, but left completeness an open problem. After nearly 40 years of active research from the process algebra community, a solution was finally found by Grabmayer [14].

A few years before this result, Grabmayer and Fokkink proved that a suitable restriction of Milner's axioms gives a complete inference system for the behaviour interpretation of a fragment of regular expressions, called the *one-free fragment* [15]. In this section, we give a quick overview of Grabmayer and Fokkink's one-free fragment [15], slightly adapted to use an alphabet that will be suitable to later use in one of the completeness proofs of skip-free GKAT.

$$\frac{}{\alpha p \xrightarrow{\alpha p} \checkmark} \qquad \frac{r_1 \xrightarrow{\alpha p} r'}{r_1 + r_2 \xrightarrow{\alpha p} r'} \qquad \frac{r_2 \xrightarrow{\alpha p} r'}{r_1 + r_2 \xrightarrow{\alpha p} r'} \qquad \frac{r_1 \xrightarrow{\alpha p} r'}{r_1 r_2 \xrightarrow{\alpha p} r' r_2}$$

$$\frac{r_1 \xrightarrow{\alpha p} \checkmark}{r_1 r_2 \xrightarrow{\alpha p} r_2} \qquad \frac{r_1 \xrightarrow{\alpha p} r'}{r_1 * r_2 \xrightarrow{\alpha p} r'(r_1 * r_2)} \qquad \frac{r_1 \xrightarrow{\alpha p} \checkmark}{r_1 * r_2 \xrightarrow{\alpha p} r_1 * r_2} \qquad \frac{r_2 \xrightarrow{\alpha p} x}{r_1 * r_2 \xrightarrow{\alpha p} x}$$

Fig. 5. The small-step semantics of one-free star expressions.

Syntax. In the process algebra literature [33,15,14], regular expressions generated by a fixed alphabet A are called *star expressions*, and denote labelled transition systems (LTSs) with labels drawn from A. As was mentioned in Section 3, skip-free automata can be seen as certain LTSs where the labels are atomic test/atomic action pairs. In Section 5, we encode skip-free GKAT expressions as one-free regular expressions and skip-free automata as LTSs with labels drawn from $\mathsf{At} \cdot \Sigma$. We instantiate the construction from [15] of the set of star expressions generated by the label set $\mathsf{At} \cdot \Sigma$.

Definition 4.1. *The set* StExp *of* one-free star expressions *is given by*

$$\mathsf{StExp} \ni r_1, r_2 ::= 0 \mid \alpha p \in \mathsf{At} \cdot \Sigma \mid r_1 + r_2 \mid r_1 r_2 \mid r_1 * r_2$$

Semantics. The semantics of StExp is now an instance of the labelled transition systems that originally appeared in [15], with atomic test/atomic action pairs as labels and a (synthetic) output state ✓ denoting successful termination.

For the rest of this paper, we call a pair (S, t) a *labelled transition system* when S is a set of states and $t \colon S \to \mathcal{P}(\mathsf{At} \cdot \Sigma \times (\checkmark + S))$ is a transition structure. We write $x \xrightarrow{\alpha p} y$ if $(\alpha p, y) \in t(x)$ and $x \xrightarrow{\alpha p} \checkmark$ if $(\alpha p, \checkmark) \in t(x)$.

The set StExp can be given the structure of a labelled transition system (StExp, τ), defined in Fig. 5. If $r \in \mathsf{StExp}$, we write $\langle r \rangle$ for the transition system obtained by restricting τ to the one-free star expressions reachable from r and call $\langle r \rangle$ the *small-step semantics* of r.

The bisimulation interpretation of one-free star expressions is subtler than the bisimulation interpretation of skip-free GKAT expressions. The issue is that labelled transition systems (LTSs) are *nondeterministic* in general: it is possible for an LTS to have both a $x \xrightarrow{\alpha p} y$ and a $x \xrightarrow{\alpha q} z$ transition for $p \neq q$ or $y \neq z$. The appropriate notion of bisimilarity for LTSs can be given as follows.

Definition 4.2. *Given labelled transition systems (S, t) and (T, u), a* bisimulation *between them is a relation $R \subseteq S \times T$ s.t. for any $x \mathrel{R} y$ and $\alpha p \in \mathsf{At} \cdot \Sigma$,*

1. *$x \xrightarrow{\alpha p} \checkmark$ if and only if $y \xrightarrow{\alpha p} \checkmark$,*
2. *if $x \xrightarrow{\alpha p} x'$, then there exist $x' \mathrel{R} y'$ such that $y \xrightarrow{\alpha p} y'$, and*
3. *if $y \xrightarrow{\alpha p} y'$, then there exist $x' \mathrel{R} y'$ such that $x \xrightarrow{\alpha p} x'$.*

As before, we denote the largest bisimulation by $\leftrightarrow$. We call x and y bisimilar *and write $x \leftrightarrow y$ if $x \mathrel{R} y$ for some bisimulation R.*

Union	Sequencing	Loops
$x = x + x$	$0x = 0$	$x * y = x(x * y) + y$
$x = x + 0$	$x(yz) = (xy)z$	$\dfrac{z = xz + y}{z = x * y}$
$x + y = y + x$	$(x + y)z = xz + yz$	
$x + (y + z) = (x + y) + z$		

Fig. 6. Axioms for equivalence for one-free star expressions.

The following closure properties of bisimulations of LTSs are useful later. They also imply that bisimilarity is an equivalence relation. Like in the skip-free case, the bisimilarity equivalence class of a state is called its *behaviour*.

Lemma 4.3. *Let* (S, t)*,* (T, u)*, and* (U, v) *be labelled transition systems. Furthermore, let* $R_1, R_2 \subseteq S \times T$ *and* $R_3 \subseteq T \times U$ *be bisimulations. Then* $R_1^{op} = \{(y, x) \mid x\ R_1\ y\}$*,* $R_1 \cup R_2$ *and* $R_1 \circ R_3$ *are bisimulations.*

Axiomatization. We follow [15], where it was shown that the axiomatization found in Fig. 6 is complete with respect to bisimilarity for one-free star expressions. Given a pair $r_1, r_2 \in \mathsf{StExp}$, we write $r_1 \equiv_* r_2$ and say that r_1 and r_2 are $\equiv_*$-*equivalent* if the equation $r_1 = r_2$ can be derived from the axioms in Fig. 6.

The following result is crucial to the next section, where we prove that the axioms of $\equiv_\dagger$ are complete with respect to bisimilarity in skip-free GKAT.

Theorem 4.4 ([15, Theorem. 7.1]). $r_1 \leftrightarrow r_2$ *if and only if* $r_1 \equiv_* r_2$*.*

5 Completeness for Skip-free Bisimulation GKAT

This section is dedicated to the proof of our first completeness result, Theorem 3.13, which says that the axioms of Fig. 2 (excluding †) are complete with respect to bisimilarity in skip-free GKAT. Our proof strategy is a reduction of our completeness result to the completeness result for StExp (Theorem 4.4).

The key objects of interest in the reduction are a pair of translations: one translation turns skip-free GKAT expressions into one-free star expressions and maintains bisimilarity, and the other translation turns (certain) one-free star expressions into skip-free GKAT expressions and maintains provable bisimilarity.

We first discuss the translation between automata and labelled transition systems, which preserves and reflects bisimilarity. We then introduce the syntactic translations and present the completeness proof.

5.1 Transforming skip-free automata to labelled transition systems

We can easily transform a skip-free automaton into an LTS by essentially turning $\xrightarrow{\alpha|p}$ transitions into $\xrightarrow{\alpha p}$ transitions. This can be formalized, as follows.

Definition 5.1. *Given a set* X*, we define* $\mathsf{grph}_X : (\bot + \Sigma \times (\checkmark + X))^{\mathsf{At}} \to \mathcal{P}(\mathsf{At} \cdot \Sigma \times (\checkmark + X))$ *to be* $\mathsf{grph}_X(\theta) = \{(\alpha p, x) \mid \theta(\alpha) = (p, x)\}$*. Given a skip-free automaton* (X, h)*, we define* $\mathsf{grph}_*(X, h) = (X, \mathsf{grph}_X \circ h)$

The function grph_X is injective: as its name suggests, $\mathsf{grph}_X(\theta)$ is essentially the graph of θ when viewed as a partial function from At to $\Sigma \times (\checkmark + X)$. This implies that the transformation grph_* of skip-free automata into LTSs preserves and reflects bisimilarity.

Lemma 5.2. *Let $x, y \in X$, and (X, h) be a skip-free automaton. Then $x \mathrel{\underline{\leftrightarrow}} y$ in (X, h) if and only if $x \mathrel{\underline{\leftrightarrow}} y$ in $\mathsf{grph}_*(X, h)$.*

Leading up to the proof of Theorem 3.13, we also need to undo the effect of grph_* on skip-free automata with a transformation that takes every LTS of the form $\mathsf{grph}_*(X, h)$ to its underlying skip-free automaton (X, h).

The LTSs that can be written in the form $\mathsf{grph}_*(X, h)$ for some skip-free automaton (X, h) can be described as follows. Call a set $U \in \mathcal{P}(\mathsf{At} \cdot \Sigma \times (\checkmark + X))$ *graph-like* if whenever $(\alpha p, x) \in U$ and $(\alpha q, y) \in U$, then $p = q$ and $x = y$. An LTS (S, t) is *deterministic* if $t(s)$ is graph-like for every $s \in S$.

Lemma 5.3. *An LTS (S, t) is deterministic if and only if $(S, t) = \mathsf{grph}_*(X, h)$ for some skip-free automaton (X, h).*

Remark 5.4. As mentioned in Footnote 7, there is a coalgebraic outlook in many of the technical details in the present paper. For the interested reader, grph and func are actually natural transformations between the functors whose coalgebras correspond to skip-free automata and labelled transitions, and are furthermore inverse to one another. This implies that grph_* and func_* witness an isomorphism between the categories of skip-free automata and deterministic LTSs.

5.2 Translating Syntax

We can mimic the transformation of skip-free automata into deterministic labelled transition systems and vice-versa by a pair of syntactic translations going back and forth between skip-free GKAT expressions and certain one-free star expressions. Similar to how only some labelled transition systems can be turned into skip-free automata, only some one-free star expressions have corresponding skip-free GKAT expressions—the deterministic ones.

The definition of deterministic expressions requires the following notation: given a test $b \in \mathsf{BExp}$, we define $b \cdot r$ inductively on $r \in \mathsf{StExp}$ as follows:

$$b \cdot 0 = 0 \qquad b \cdot \alpha p = \begin{cases} \alpha p & \alpha \leq b \\ 0 & \alpha \not\leq b \end{cases} \qquad b \cdot (r_1 + r_2) = b \cdot r_1 + b \cdot r_2$$
$$b \cdot (r_1 r_2) = (b \cdot r_1) r_2 \qquad b \cdot (r_1 * r_2) = (b \cdot r_1)(r_1 * r_2) + b \cdot r_2$$

for any $\alpha p \in \mathsf{At} \cdot \Sigma$ and $r_1, r_2 \in \mathsf{StExp}$.

Definition 5.5. *The set of* deterministic *one-free star expressions is the smallest subset $\mathsf{Det} \subseteq \mathsf{StExp}$ such that $0 \in \mathsf{Det}$ and $\alpha p \in \mathsf{Det}$ for any $\alpha \in \mathsf{At}$ and $p \in \Sigma$, and for any $r_1, r_2 \in \mathsf{Det}$, and $b \in \mathsf{BExp}$, $b \cdot r_1 + \bar{b} \cdot r_2, r_1 r_2$, and $(b \cdot r_1) * (\bar{b} \cdot r_2) \in \mathsf{Det}$.*

From GExp^- to Det. We can now present the translations of skip-free expressions to deterministic one-free star expressions.

Definition 5.6. *We define the* translation *function* $\mathrm{gtr} : \mathsf{GExp}^- \to \mathsf{Det}$ *by*

$$\mathrm{gtr}(0) = 0 \qquad \mathrm{gtr}(p) = \sum_{\alpha \in \mathsf{At}} \alpha p \qquad \mathrm{gtr}(e_1 +_b e_2) = b \cdot \mathrm{gtr}(e_1) + \bar{b} \cdot \mathrm{gtr}(e_2)$$
$$\mathrm{gtr}(e_1 \cdot e_2) = \mathrm{gtr}(e_1)\,\mathrm{gtr}(e_2) \qquad \mathrm{gtr}(e_1^{(b)} e_2) = (b \cdot e_1) * (\bar{b} \cdot e_2)$$

for any $b \in \mathsf{BExp}$*,* $p \in \Sigma$*,* $e_1, e_2 \in \mathsf{GExp}$*.*

Remark 5.7. In Definition 5.6, we make use of a generalized sum $\sum_{\alpha \in \mathsf{At}}$. Technically, this requires we fix an enumeration of At ahead of time, say $\mathsf{At} = \{\alpha_1, \dots, \alpha_n\}$, at which point we can define $\sum_{\alpha \in \mathsf{At}} r_\alpha = r_{\alpha_1} + \cdots + r_{\alpha_n}$. Of course, $+$ is commutative and associative up to $\equiv_*$, so the actual ordering of this sum does not matter as far as equivalence is concerned.

The most prescient feature of this translation is that it respects bisimilarity.

Lemma 5.8. *The graph of the translation function* gtr *is a bisimulation of labelled transition systems between* $\mathsf{grph}_*(\mathsf{GExp}^-, \partial)$ *and* (StExp, τ)*. Consequently, if* $e_1 \underline{\leftrightarrow} e_2$ *in* $\mathsf{grph}_*(\mathsf{GExp}^-, \partial)$*, then* $\mathrm{gtr}(e_1) \underline{\leftrightarrow} \mathrm{gtr}(e_2)$ *in* (StExp, τ)*.*

From Det to GExp^-. We would now like to define a *back translation* function $\mathrm{rtg} : \mathsf{Det} \to \mathsf{GExp}^-$ by induction on its argument. Looking at Definition 5.5, one might be tempted to write $\mathrm{rtg}(b \cdot r_1 + \bar{b} \cdot r_2) = \mathrm{rtg}(r_1) +_b \mathrm{rtg}(r_2)$, but the fact of the matter is that it is possible for there to be distinct $b, c \in \mathsf{BExp}$ such that $b \cdot r_1 + \bar{b} \cdot r_2 = c \cdot r_1 + \bar{c} \cdot r_2$, even when b and c have different atoms.

Definition 5.9. *Say that* $r_1, r_2 \in \mathsf{StExp}$ *are* separated by $b \in \mathsf{BExp}$ *if* $r_1 = b \cdot r_1$ *and* $r_2 = \bar{b} \cdot r_2$*. If such a* b *exists we say that* r_1 *and* r_2 *are* separated.

Another way to define Det is therefore to say that Det is the smallest subset of StExp containing 0 and $\mathsf{At} \cdot \Sigma$ that is closed under sequential composition and closed under unions and stars of separated one-free star expressions.

Suppose r_1 and r_2 are separated by both b and c. Then one can prove that $(b \vee c)r_1 \equiv_* br_1 + cr_1 \equiv_* r_1$ and $\overline{(b \vee c)}r_2 = (\bar{b} \wedge \bar{c})r_2 \equiv_* \bar{b}(\bar{c}r_2) \equiv_* r_2$, so r_1 and r_2 are separated by $b \vee c$ as well. Since there are only finitely many Boolean expressions up to equivalence, there is a maximal (weakest) test $b(r_1, r_2) \in \mathsf{BExp}$ such that r_1 and r_2 are separated by $b(r_1, r_2)$.

Definition 5.10. *The* back translation $\mathrm{rtg} : \mathsf{Det} \to \mathsf{GExp}^-$ *is defined by*

$$\mathrm{rtg}(0) = 0 \qquad \mathrm{rtg}(\alpha p) = p +_\alpha 0 \qquad \mathrm{rtg}(r_1 + r_2) = \mathrm{rtg}(r_1) +_{b(r_1, r_2)} \mathrm{rtg}(r_2)$$
$$\mathrm{rtg}(r_1 r_2) = \mathrm{rtg}(r_1) \cdot \mathrm{rtg}(r_2) \qquad \mathrm{rtg}(r_1 * r_2) = \mathrm{rtg}(r_1)^{(b(r_1, r_2))}\,\mathrm{rtg}(r_2)$$

for any $r_1, r_2 \in \mathsf{StExp}$*. In the union and star cases, we may use that* r_1 *and* r_2 *are separated (by definition of* Det*), so that* $b(r_1, r_2)$ *is well-defined.*

The most prescient property of rtg is that it preserves provable equivalence.

Lemma 5.11. *Let* $r_1, r_2 \in \mathsf{Det}$. *If* $r_1 \equiv_* r_2$, *then* $\mathrm{rtg}(r_1) \equiv_\dagger \mathrm{rtg}(r_2)$.

The last fact needed in the proof of completeness is that, up to provable equivalence, every skip-free GKAT expression is equivalent to its back-translation.

Lemma 5.12. *For any* $e \in \mathsf{GExp}^-$, $e \equiv_\dagger \mathrm{rtg}(\mathrm{gtr}(e))$.

We are now ready to prove Theorem 3.13, that provable bisimilarity is complete with respect to behavioural equivalence in skip-free GKAT.

Theorem 3.13 (Completeness I). *If* $e_1 \underline{\leftrightarrow} e_2$, *then* $e_1 \equiv_\dagger e_2$.

Proof. Let $e_1, e_2 \in \mathsf{GExp}$ be a bisimilar pair of skip-free GKAT expressions. By Lemma 5.2, e_1 and e_2 are bisimilar in $\mathsf{grph}_*(\mathsf{GExp}^-, \partial)$. By Lemmas 4.3 and 5.8, the translation $\mathrm{gtr} : \mathsf{grph}_*(\mathsf{GExp}^-, \partial) \to (\mathsf{StExp}, \tau)$ preserves bisimilarity, so $\mathrm{gtr}(e_1)$ and $\mathrm{gtr}(e_2)$ are bisimilar in (StExp, τ) as well. By Theorem 4.4, $\mathrm{gtr}(e_1) \equiv_* \mathrm{gtr}(e_2)$. Therefore, by Lemma 5.11, $\mathrm{rtg}(\mathrm{gtr}(e_1)) \equiv_\dagger \mathrm{rtg}(\mathrm{gtr}(e_2))$. Finally, by Lemma 5.12, we have $e_1 \equiv_\dagger \mathrm{rtg}(\mathrm{gtr}(e_1)) \equiv_\dagger \mathrm{rtg}(\mathrm{gtr}(e_2)) \equiv_\dagger e_2$.

6 Completeness for Skip-free GKAT

The previous section establishes that $\equiv_\dagger$-equivalence coincides with bisimilarity for skip-free GKAT expressions by reducing the completeness problem of skip-free GKAT up to bisimilarity to a solved completeness problem, namely that of one-free star expressions up to bisimilarity. In this section we prove a completeness result for skip-free GKAT up to language equivalence. We show this can be achieved by reducing it to the completeness problem of skip-free GKAT up to bisimilarity, which we just solved in the previous section.

Despite bisimilarity being a less traditional equivalence in the context of Kleene algebra, this reduction simplifies the completeness proof greatly, and justifies the study of bisimilarity in the pursuit of completeness for GKAT.

The axiom $x \cdot 0 = 0$ (which is the only difference between skip-free GKAT up to language equivalence and skip-free GKAT up to bisimilarity) indicates that the only semantic difference between bisimilarity and language equivalence in skip-free GKAT is early termination. This motivates our reduction to skip-free GKAT up to bisimilarity below, which involves reducing each skip-free expression to an expression representing only the successfully terminating branches of execution.

Now let us turn to the formal proof of Theorem 3.14, which says that if $e, f \in \mathsf{GExp}^-$ are such that $\mathcal{L}(e) = \mathcal{L}(f)$, then $e \equiv f$. In a nutshell, our strategy is to produce two terms $\lfloor e \rfloor, \lfloor f \rfloor \in \mathsf{GExp}^-$ such that $e \equiv \lfloor e \rfloor$, $f \equiv \lfloor f \rfloor$ and $\lfloor e \rfloor \underline{\leftrightarrow} \lfloor f \rfloor$ in $(\mathsf{GExp}^-, \partial)$. The latter property tells us that $\lfloor e \rfloor \equiv_\dagger \lfloor f \rfloor$ by Theorem 3.13, which allows us to conclude $e \equiv f$. The expression $\lfloor e \rfloor$ can be thought of as the *early termination version* of e, obtained by pruning the branches of its execution that cannot end in successful termination.

To properly define the transformation $\lfloor - \rfloor$ on expressions, we need the notion of a *dead* state in a skip-free automaton, analogous to a similar notion from [42].

Definition 6.1. *Let (X, h) be a skip-free automaton. The set $D(X, h)$ is the largest subset of X such for all $x \in D(X, h)$ and $\alpha \in \mathsf{At}$, either $h(x)(\alpha) = \bot$ or $h(x)(\alpha) \in \Sigma \times D(X, h)$. When $x \in D(X, h)$, x is* dead*; otherwise, it is* live*.*

In the sequel, we say $e \in \mathsf{GExp}^-$ is dead when e is a dead state in $(\mathsf{GExp}^-, \partial)$, i.e., when $e \in D(\mathsf{GExp}^-, \partial)$. Whether e is dead can be determined by a simple depth-first search, since e can reach only finitely many expressions by ∂. The axioms of skip-free GKAT can also tell when a skip-free expression is dead.

Lemma 6.2. *Let $e \in \mathsf{GExp}$. If e is dead, then $e \equiv 0$.*

We are now ready to define $\lfloor - \rfloor$, the transformation on expressions promised above. The intuition here is to prune the dead subterms of e by recursive descent; whenever we find a part that will inevitably lead to an expression that is never going to lead to acceptance, we set it to 0.

Definition 6.3. *Let $e \in \mathsf{GExp}^-$ and $a \in \mathsf{BExp}$. In the sequel we use ae as a shorthand for $e +_a 0$. We furthermore define $\lfloor e \rfloor$ inductively, as follows*

$$\lfloor 0 \rfloor = 0 \qquad \lfloor p \rfloor = p \qquad \lfloor e_1 +_b e_2 \rfloor = \lfloor e_1 \rfloor +_b \lfloor e_2 \rfloor$$

$$\lfloor e_1 \cdot e_2 \rfloor = \begin{cases} 0 & e_2 \text{ is dead} \\ \lfloor e_1 \rfloor \cdot \lfloor e_2 \rfloor & \text{otherwise} \end{cases} \qquad \lfloor e_1{}^{(b)} e_2 \rfloor = \begin{cases} 0 & \bar{b} e_2 \text{ is dead} \\ \lfloor e_1 \rfloor^{(b)} \lfloor e_2 \rfloor & \text{otherwise} \end{cases}$$

The transformation defined above yields a term that is $\equiv$-equivalent to e, provided that we include the early termination axiom $e \cdot 0 \equiv 0$. The proof is a simple induction on e, using Lemma 6.2.

Lemma 6.4. *For any $e \in \mathsf{GExp}^-$, $e \equiv \lfloor e \rfloor$.*

It remains to show that if $\mathcal{L}(e) = \mathcal{L}(f)$, then $\lfloor e \rfloor$ and $\lfloor f \rfloor$ are bisimilar. To this end, we need to relate the language semantics of e and f to their behaviour. As a first step, we note that behaviour that never leads to acceptance can be pruned from a skip-free automaton by removing transitions into dead states.

Definition 6.5. *Let (X, h) be a skip-free automaton. Define $\lfloor h \rfloor : X \to GX$ by*

$$\lfloor h \rfloor(x)(\alpha) = \begin{cases} \bot & h(x)(\alpha) = (p, x'),\ x' \text{ is dead} \\ h(x)(\alpha) & \text{otherwise} \end{cases}$$

Moreover, language equivalence of two states in a skip-free automaton implies bisimilarity of those states, but only in the pruned version of that skip-free automaton. The proof works by showing that the relation on X that connects states with the same language is, in fact, a bisimulation in $(X, \lfloor h \rfloor)$.

Lemma 6.6. *Let (X, h) be a skip-free automaton and $x, y \in X$. We have*

$$\mathcal{L}(x, (X, h)) = \mathcal{L}(y, (X, h)) \implies x \leftrightarroweq y \text{ in } (X, \lfloor h \rfloor)$$

The final intermediate property relates the behaviour of to states in the pruned skip-free automaton of expressions to the syntactic skip-free automaton.

Lemma 6.7. *The graph* $\{(e, \lfloor e \rfloor) \mid e \in \mathsf{GExp}^-\}$ *of* $\lfloor - \rfloor$ *is a bisimulation of skip-free automata between* $(\mathsf{GExp}^-, \lfloor \partial \rfloor)$ *and* $(\mathsf{GExp}^-, \partial)$.

We now have all the ingredients necessary to prove Theorem 3.14.

Theorem 3.14 (Completeness II). *If* $e_1 \sim_{\mathcal{L}} e_2$*, then* $e_1 \equiv e_2$.

Proof. If $e_1 \sim_{\mathcal{L}} e_2$, then by definition $\mathcal{L}(e_1) = \mathcal{L}(e_2)$. By Lemma 6.6, $e_1 \leftrightarrow e_2$ in $(\mathsf{GExp}^-, \lfloor \partial \rfloor)$, which by Lemma 6.7 implies that $\lfloor e_1 \rfloor \leftrightarrow \lfloor e_2 \rfloor$ in $(\mathsf{GExp}^-, \partial)$. From Theorem 3.13 we know that $\lfloor e_1 \rfloor \equiv_\dagger \lfloor e_2 \rfloor$, and therefore $e_1 \equiv e_2$ by Lemma 6.4.

7 Relation to GKAT

So far we have seen the technical development of skip-free GKAT without much reference to the original development of GKAT as it was presented in [42] and [40]. In this section, we make the case that the semantics of skip-free GKAT is merely a simplified version of the semantics of GKAT, and that the two agree on which expressions are equivalent after embedding skip-free GKAT into GKAT. More precisely, we identify the bisimulation and language semantics of skip-free GKAT given in Section 3 with instances of the existing bisimulation [40] and language [42] semantics of GKAT proper. The main takeaway is that two skip-free GKAT expressions are equivalent in our semantics precisely when they are equivalent when interpreted as proper GKAT expressions in the existing semantics.

7.1 Bisimulation semantics

To connect the bisimulation semantics of skip-free GKAT to GKAT at large, we start by recalling the latter. To do this, we need to define GKAT automata.

Definition 7.1. *A (GKAT)* automaton *is a pair* (X, d) *such that* X *is a set and* $d : X \to (\bot + \checkmark + \Sigma \times X)^{\mathsf{At}}$ *is a function called the* transition function. *We write* $x \xrightarrow{\alpha|p} y$ *to denote* $d(x)(\alpha) = (p, y)$, $x \Rightarrow \alpha$ *to denote* $d(x)(\alpha) = \checkmark$, *and* $x \downarrow \alpha$ *if* $d(x)(\alpha)$ *is undefined.*

Automata can be equipped with their own notion of bisimulation.[8]

Definition 7.2. *Given automata* (X, h) *and* (Y, k), *a* bisimulation *between them is a relation* $R \subseteq X \times Y$ *such that if* $x \mathrel{R} y$, $\alpha \in \mathsf{At}$ *and* $p \in \Sigma$,:

1. *if* $h(x)(\alpha) = \bot$, *then* $k(y)(\alpha) = \bot$; *and*
2. *if* $h(x)(\alpha) = \checkmark$, *then* $k(y)(\alpha) = \checkmark$; *and*
3. *if* $h(x)(\alpha) = (p, x')$, *then* $k(y)(\alpha) = (p, y')$ *such that* $x' \mathrel{R} y'$.

[8] As in previous sections, automata can be studied as coalgebras for a given functor and the notions below are instances of general abstract notions [17,37].

$$\frac{\alpha \leq b}{b \Rightarrow a} \quad \frac{\alpha \leq b \quad e_1 \Rightarrow \alpha}{e_1 +_b e_2 \Rightarrow \alpha} \quad \frac{\alpha \leq \bar{b} \quad e_2 \Rightarrow \alpha}{e_1 +_b e_2 \Rightarrow a} \quad \frac{\alpha \leq b \quad e_1 \xrightarrow{\alpha|p} e'}{e_1 +_b e_2 \xrightarrow{\alpha|p} e'} \quad \frac{\alpha \leq \bar{b} \quad e_2 \xrightarrow{\alpha|p} e'}{e_1 +_b e_2 \xrightarrow{\alpha|p} e'}$$

$$\frac{}{p \xrightarrow{\alpha|p} 1} \quad \frac{e \Rightarrow \alpha \quad e_2 \Rightarrow \alpha}{e_1 \cdot e_2 \Rightarrow a} \quad \frac{e \Rightarrow \alpha \quad f \xrightarrow{\alpha|p} e'}{e_1 \cdot e_2 \xrightarrow{\alpha|p} e'} \quad \frac{e \xrightarrow{\alpha|p} e'}{e_1 \cdot e_2 \xrightarrow{\alpha|p} e' \fatsemi e_2}$$

$$\frac{\alpha \leq b \quad e \xrightarrow{\alpha|p} e'}{e^{(b)} \xrightarrow{\alpha|p} e' \fatsemi e^{(b)}} \quad \frac{\alpha \leq \bar{b}}{e^{(b)} \Rightarrow a}$$

Fig. 7. The transition function $\delta : \mathsf{GExp} \to (\bot + \checkmark + \Sigma \times \mathsf{GExp})^{\mathsf{At}}$ defined inductively. Here, $e_1 \fatsemi e_2$ is e_2 when $e = 1$ and $e_1 \cdot e_2$ otherwise, $b \in \mathsf{BExp}$, $p \in \Sigma$, and $e, e', e_i \in \mathsf{GExp}$.

We call x and y bisimilar *and write $x \leftrightarroweq y$ if $x \mathrel{R} y$ for some bisimulation R.*

Remark 7.3. The properties listed above are implications, but it is not hard to show that if all three properties hold for R, then so do all of their symmetric counterparts. For instance, if $k(y)(\alpha) = (p, y')$, then certainly $h(x)(\alpha)$ must be of the form (q, x'), which then implies that $q = p$ while $x' \mathrel{R} y'$.

Two GKAT expressions are bisimilar when they are bisimilar as states in the *syntactic automaton* [40], (GExp, δ), summarised in Fig. 7.

Remark 7.4. The definition of δ given above diverges slightly from the definition in [40]. Fortunately, this does not make a difference in terms of the bisimulation semantics: two expressions are bisimilar in (GExp, δ) if and only if they are bisimilar in the original semantics. The full version [22] contains a detailed account.

There is a fairly easy way to convert a skip-free automaton into a GKAT automaton: simply reroute all accepting transitions into a new state $\top$, that accepts immediately, and leave the other transitions the same.

Definition 7.5. *Given a skip-free automaton (X, d), we define the automaton* $\mathsf{embed}(X, d) = (X + \top, \tilde{d})$, *where $\tilde{d}$ is defined by*

$$\tilde{d}(x)(\alpha) = \begin{cases} \checkmark & x = \top \\ (p, \top) & d(x)(\alpha) = (p, \checkmark) \\ d(x)(\alpha) & \textit{otherwise} \end{cases}$$

We can show that two states are bisimilar in a skip-free automaton if and only if these same states are bisimilar in the corresponding GKAT automaton.

Lemma 7.6. *Let (X, d) be a skip-free automaton, and let $x, y \in X$.*

$$x \leftrightarroweq y \textit{ in } (X, d) \iff x \leftrightarroweq y \textit{ in } \mathsf{embed}(X, d)$$

The syntactic skip-free automaton $(\mathsf{GExp}^-, \partial)$ can of course be converted to a GKAT automaton in this way. It turns out that there is a very natural way of correlating this automaton to the syntactic GKAT automaton (GExp, δ).

Lemma 7.7. *The relation* $\{(e, e) : e \in \mathsf{GExp}^-\} \cup \{(\top, 1)\}$ *is a bisimulation between* $\mathsf{embed}(\mathsf{GExp}^-, \partial)$ *and* (GExp, δ).

We now have everything to relate the bisimulation semantics of skip-free GKAT expressions to the bisimulation semantics of GKAT expressions at large.

Lemma 7.8. *Let* $e, f \in \mathsf{GExp}^-$. *The following holds:*

$$e \,\underline{\leftrightarrow}\, f \textit{ in } (\mathsf{GExp}^-, \partial) \iff e \,\underline{\leftrightarrow}\, f \textit{ in } (\mathsf{GExp}, \delta)$$

Proof. We derive using Lemmas 7.6 and 7.7, as follows: since the graph of embed is a bisimulation, $e \,\underline{\leftrightarrow}\, f$ in $(\mathsf{GExp}^-, \partial)$ iff $e \,\underline{\leftrightarrow}\, f$ in $\mathsf{embed}(\mathsf{GExp}^-, \partial)$ if and only if $e \,\underline{\leftrightarrow}\, f$ in (GExp, δ). In the last step, we use the fact that if R is a bisimulation (of automata) between (X, h) and (Y, k), and S is a bisimulation between (Y, k) and (Z, ℓ), then $R \circ S$ is a bisimulation between (X, h) and (Z, ℓ).

7.2 Language semantics

We now recall the language semantics of GKAT, which is defined in terms of *guarded strings* [29], i.e., words in the set $\mathsf{At} \cdot (\Sigma \cdot \mathsf{At})^*$, where atoms and actions alternate. In GKAT, successful termination occurs with a trailing associated test, representing the state of the machine at termination. In an execution of the sequential composition of two programs $e \cdot f$, the test trailing the execution of e needs to match up with an input test compatible with f, otherwise the program crashes at the end of executing e. The following operations on languages of guarded strings record this behaviour by matching the ends of traces on the left with the beginnings of traces on the right.

Definition 7.9. *For* $L, K \subseteq At \cdot (\Sigma \cdot \mathsf{At})^*$, *define* $L \diamond K = \{w\alpha x : w\alpha \in L, \alpha x \in K\}$ *and* $L^{(*)} = \bigcup_{n \in \mathbb{N}} L^{(n)}$, *where* $L^{(n)}$ *is defined inductively by setting* $L^{(0)} = \mathsf{At}$ *and* $L^{(n+1)} = L \diamond L^{(n)}$.

The language semantics of a GKAT expression is now defined in terms of the composition operators above, as follows.

Definition 7.10. *We define* $\widehat{\mathcal{L}} : \mathsf{GExp} \to \mathcal{P}(\mathsf{At} \cdot (\Sigma \cdot \mathsf{At})^*)$ *inductively, as follows:*

$$\widehat{\mathcal{L}}(b) = \{\alpha \in \mathsf{At} \mid \alpha \leq b\} \qquad \widehat{\mathcal{L}}(p) = \{\alpha p \beta \mid \alpha, \beta \in \mathsf{At}\} \qquad \widehat{\mathcal{L}}(e \cdot f) = \widehat{\mathcal{L}}(e) \diamond \widehat{\mathcal{L}}(f)$$

$$\widehat{\mathcal{L}}(e +_b f) = \widehat{\mathcal{L}}(b) \diamond \widehat{\mathcal{L}}(e) \cup \widehat{\mathcal{L}}(\bar{b}) \diamond \widehat{\mathcal{L}}(f) \qquad \widehat{\mathcal{L}}(e^{(b)}) = (\widehat{\mathcal{L}}(b) \diamond \widehat{\mathcal{L}}(e))^{(*)} \diamond \widehat{\mathcal{L}}(\bar{b})$$

This semantics is connected to the relational semantics from Definition 2.2:

Theorem 7.11 ([42]). *For* $e, f \in \mathsf{GExp}$, *we have* $\widehat{\mathcal{L}}(e) = \widehat{\mathcal{L}}(f)$ *if and only if* $[\![e]\!]_\sigma = [\![f]\!]_\sigma$ *for all relational interpretations* σ

Moreover, since skip-free GKAT expressions are also GKAT expressions, this means that we now have two language interpretations of the former, given by $\widehat{\mathcal{L}}$ and $\mathcal{L}$. Fortunately, one can easily be expressed in terms of the other.

Guarded Union	**Sequencing**	**Loops**
$x = x +_b x$	$x(yz) = (xy)z$	$xx^{(b)} +_b 1 = x^{(b)}$
$x +_b y = y +_{\bar{b}} x$	$0x = 0$	$(x +_a 1)^{(b)} = (ax)^{(b)}$
$x +_b (y +_c z) = (x +_b y) +_{b \vee c} z$	$x0 \overset{(\dagger)}{=} 0$	$\dfrac{z = xz +_b y \quad E(x) = 0}{z = x^{(b)}y}$
$x +_b y = bx +_b y$	$1x = x$	
$(x +_b y)z = xz +_b yz$	$x1 = x$	

Fig. 8. Axioms for language semantics GKAT (without the Boolean algebra axioms for tests). The function $E : \mathsf{GExp} \to \mathsf{BExp}$ is defined below. If the axiom marked (†) is omitted, the above potentially axiomatizes bisimilarity.

Lemma 7.12. *For $e \in \mathsf{GExp}^-$, it holds that $\widehat{\mathcal{L}}(e) = \mathcal{L}(e) \cdot \mathsf{At}$.*

As an easy consequence of the above, we find that the two semantics must identify the same skip-free GKAT-expressions.

Lemma 7.13. *For $e, f \in \mathsf{GExp}^-$, we have $\mathcal{L}(e) = \mathcal{L}(f)$ iff $\widehat{\mathcal{L}}(e) = \widehat{\mathcal{L}}(f)$.*

By Theorem 3.14, these properties imply that $\equiv$ also axiomatizes relational equivalence of skip-free GKAT-expressions, as a result.

Corollary 7.14. *Let $e, f \in \mathsf{GExp}^-$, we have $e \equiv f$ if and only if $[\![e]\!]_\sigma = [\![f]\!]_\sigma$ for all relational interpretations σ.*

7.3 Equivalences

Finally, we relate equivalences as proved for skip-free GKAT expressions to those provable for GKAT expressions, showing that proofs of equivalence for skip-free GKAT expressions can be replayed in the larger calculus, without (UA).

The axioms of GKAT as presented in [42,40] are provided in Figure 8. We write $e \approx_\dagger f$ when $e = f$ is derivable from the axioms in Figure 8 with the exception of (†), and $e \approx f$ when $e = f$ is derivable from the full set.

The last axiom of GKAT is not really a single axiom, but rather an *axiom scheme*, parameterized by the function $E : \mathsf{GExp} \to \mathsf{BExp}$ defined as follows:

$$E(b) = b \qquad E(p) = 0 \qquad E(e +_b f) = (b \wedge E(e)) \vee (\bar{b} \wedge E(f))$$
$$E(e \cdot f) = E(e) \wedge E(f) \qquad E(e^{(b)}) = \bar{b}$$

The function E models the analogue of Salomaa's *empty word property* [38]: we say e is *guarded* when $E(b)$ is equivalent to 0 by to the laws of Boolean algebra. Notice that as GKAT expressions, skip-free GKAT expressions are always guarded.

Since skip-free GKAT expressions are also GKAT expressions, we have four notions of equivalence for GKAT expressions: as skip-free expressions or GKAT expressions in general, either with or without (†). These are related as follows.

Theorem 7.15. *Let $e, f \in \mathsf{GExp}^-$. Then (1) $e \approx_\dagger f$ if and only if $e \equiv_\dagger f$, and (2) $e \approx f$ if and only if $e \equiv f$.*

Proof. For the forward direction of (1), we note that if $e \approx_\dagger f$, then $e \leftrightarrow f$ in (GExp, δ) by Theorem 3.12. By Lemma 7.8, $e \leftrightarrow f$ in $(\mathsf{GExp}^-, \delta)$ and therefore $e \equiv_\dagger f$ by Theorem 3.13. Conversely, note that any proof of $e = f$ by the axioms of Figure 2 can be replayed using the rules from Figure 8. In particular, the guardedness condition required for the last skip-free GKAT axiom using the last GKAT axiom is always true, because $E(g) \approx_\dagger 0$ for any $g \in \mathsf{GExp}^-$.

The proof of the second claim is similar, but uses Theorem 3.13 instead.

8 Related Work

This paper fits into a larger research program focused on understanding the logical and algebraic content of programming. Kleene's paper introducing the algebra of regular languages [23] was a foundational contribution to this research program, containing an algebraic account of mechanical programming and some of its sound equational laws. The paper also contained an interesting completeness problem: give a complete description of the equations satisfied by the algebra of regular languages. Salomaa was the first to provide a sound and complete axiomatization of language equivalence for regular expressions [38].

The axiomatization in op. cit. included an inference rule with a side condition that prevented it from being *algebraic* in the sense that the validity of an equation is not preserved when substituting letters for arbitrary regular expressions. Nevertheless, this inspired axiomatizations of several variations and extensions of Kleene algebra [46,42,41], as well as Milner's axiomatization of the algebra of star behaviours [33]. The side condition introduced by Salomaa is often called the *empty word property*, an early version of a concept from process theory called *guardedness*[9] that is also fundamental to the theory of iteration [6].

Our axiomatization of skip-free GKAT *is* algebraic due to the lack of a guardedness side-condition (it is an equational *Horn theory* [32]). This is particularly desirable because it allows for an abundance of other models of the axioms. Kozen proposed an algebraic axiomatization of Kleene algebra that is sound and complete for language equivalence [25], which has become the basis for a number of axiomatizations of other Kleene algebra variants [13,19,20,47] including Kleene algebra with tests [26]. KAT also has a plethora of relational models, which are desirable for reasons we hinted at in Section 2.

GKAT is a fragment of KAT that was first identified in [30]. It was later given a sound and complete axiomatization in [42], although the axiomatization is neither algebraic nor finite (it includes (UA), an axiom scheme that stands for infinitely many axioms). It was later shown that dropping $x \cdot 0 = 0$ (called (S3) in [42]) from this axiomatization gives a sound and complete axiomatization of bisimilarity [40]. The inspiration for our pruning technique is also in [40], where a reduction of the language equivalence case to the bisimilarity case is discussed.

[9] This is a different use of the word "guarded" than in "guarded Kleene algebra with tests". In the context of process theory, a recursive specification is guarded if every of its function calls occurs within the scope of an operation.

Despite the existence of an algebraic axiomatization of language equivalence in KAT, GKAT has resisted algebraic axiomatization so far. Skip-free GKAT happens to be a fragment of GKAT in which every expression is guarded, thus eliminating the need for the side condition in Fig. 8 and allowing for an algebraic axiomatization. An inequational axiomatization resembling that of KAT might be gleaned from the recent preprint [39], but we have not investigated this carefully. The GKAT axioms for bisimilarity of ground terms can also likely be obtained from the small-step semantics of GKAT using [1,2,3], but unfortunately this does not appear to help with the larger completeness problem.

The idea of reducing one completeness problem in Kleene algebra to another is common in Kleene algebra; for instance, it is behind the completeness proof of KAT [29]. Cohen also reduced *weak Kleene algebra* as an axiomatization of star expressions up to simulation to *monodic trees* [10], whose completeness was conjectured by Takai and Furusawa [45]. Grabmayer's solution to the completeness problem of regular expressions modulo bisimulation [14] can also be seen as a reduction to the one-free case [15], since his *crystallization* procedure produces an automaton that can be solved using the technique found in op. cit. Other instances of reductions include [9,4,11,47,19,21,31,35,27]. Recent work has started to study reductions and their compositionality properties [11,20,34].

9 Discussion

We continue the study of efficient fragments of Kleene Algebra with Tests (KAT) initiated in [42], where the authors introduced Guarded Kleene Algebra with Tests (GKAT) and provided an efficient decision procedure for equivalence. They also proposed a candidate axiomatization, but left open two questions.

- The first question concerned the existence of an algebraic axiomatization, which is an axiomatization that is closed under substitution—i.e., where one can prove properties about a certain program p and then use p as a variable in the context of a larger program, being able to substitute as needed. This is essential to enable compositional analysis.
- The second question left open in [42] was whether an axiomatization that did not require an axiom scheme was possible. Having a completeness proof that does not require an axiom scheme to reason about mutually dependent loops is again essential for scalability: we should be able to axiomatize single loops and generalize this behaviour to multiple, potentially, nested loops.

In this paper, we identified a large fragment of GKAT, which we call *skip-free GKAT* (GKAT^-), that can be axiomatized algebraically without relying on an axiom scheme. We show how the axiomatization works well for two types of equivalence: bisimilarity and language equivalence, by proving completeness results for both semantics. Having the two semantics is interesting from a verification point of view as it gives access to different levels of precision when analyzing program behaviour, but also enables a layered approach to the completeness proofs.

We provide a reduction of the completeness proof for language semantics to the one for bisimilarity. Moreover, the latter is connected to a recently solved [14] problem proposed by Milner. This approach enabled two things: it breaks down the completeness proofs and reuses some of the techniques while also highlighting the exact difference between the two equivalences (captured by the axiom $e \cdot 0 \equiv 0$ which does not hold for bisimilarity). We also showed that proofs of equivalence in skip-free GKAT transfer without any loss to proofs of equivalence in GKAT.

There are several directions for future work. The bridge between process algebra and Kleene algebra has not been exploited to its full potential. The fact that we could reuse results by Grabmayer and Fokkink [14,15] was a major step towards completeness. An independent proof would have been much more complex and very likely required the development of technical tools resembling those in [14,15]. We hope the results in this paper can be taken further and more results can be exchanged between the two communities to solve open problems.

The completeness problem for full GKAT remains open, but our completeness results for skip-free GKAT are encouraging. We believe they show a path towards studying whether an algebraic axiomatization can be devised or a negative result can be proved. A first step in exploring a completeness result would be to try extending Grabmayer's completeness result [14] to a setting with output variables—this is a non-trivial exploration, but we are hopeful will yield new tools for completeness. As mentioned in the introduction, NetKAT [4] (and its probabilistic variants [12,43]) have been one of the most successful extensions of KAT. We believe the step from skip-free GKAT to a skip-free guarded version of NetKAT is also a worthwhile exploration. Following [16], we hope to be able to explore these extensions in a modular and parametric way.

Acknowledgements A. Silva and T. Schmid were partially funded by ERC grant Autoprobe (grant agreement 101002697). T. Kappé was supported by the EU's Horizon 2020 research and innovation programme under Marie Skłodowska-Curie grant agreement No. 101027412 (VERLAN).

References

1. Aceto, L.: Deriving complete inference systems for a class of GSOS languages generation regular behaviours. In: CONCUR. pp. 449–464 (1994). https://doi.org/10.1007/978-3-540-48654-1_33
2. Aceto, L., Caltais, G., Goriac, E., Ingólfsdóttir, A.: Axiomatizing GSOS with predicates. In: SOS. pp. 1–15 (2011). https://doi.org/10.4204/EPTCS.62.1
3. Aceto, L., Caltais, G., Goriac, E., Ingólfsdóttir, A.: PREG axiomatizer - A ground bisimilarity checker for GSOS with predicates. In: CALCO. pp. 378–385 (2011). https://doi.org/10.1007/978-3-642-22944-2_27
4. Anderson, C.J., Foster, N., Guha, A., Jeannin, J.B., Kozen, D., Schlesinger, C., Walker, D.: NetKAT: semantic foundations for networks. In: POPL. pp. 113–126 (2014). https://doi.org/10.1145/2535838.2535862
5. Birkhoff, G.: On the structure of abstract algebras. Mathematical Proceedings of the Cambridge Philosophical Society **31**(4), 433–454 (1935). https://doi.org/10.1017/S0305004100013463

6. Bloom, S.L., Ésik, Z.: Iteration Theories - The Equational Logic of Iterative Processes. EATCS Monographs on Theoretical Computer Science, Springer (1993). https://doi.org/10.1007/978-3-642-78034-9
7. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM **11**(4), 481–494 (1964). https://doi.org/10.1145/321239.321249
8. Chajed, T., Tassarotti, J., Kaashoek, M.F., Zeldovich, N.: Argosy: verifying layered storage systems with recovery refinement. In: PLDI. pp. 1054–1068 (2019). https://doi.org/10.1145/3314221.3314585
9. Cohen, E.: Hypotheses in Kleene algebra. Tech. rep., Bellcore (1994)
10. Cohen, E.: Weak Kleene algebra is sound and (possibly) complete for simulation (2009). https://doi.org/10.48550/arXiv.0910.1028
11. Doumane, A., Kuperberg, D., Pous, D., Pradic, P.: Kleene algebra with hypotheses. In: FOSSACS. pp. 207–223 (2019). https://doi.org/10.1007/978-3-030-17127-8_12
12. Foster, N., Kozen, D., Mamouras, K., Reitblatt, M., Silva, A.: Probabilistic NetKAT. In: ESOP. pp. 282–309 (2016). https://doi.org/10.1007/978-3-662-49498-1_12
13. Foster, N., Kozen, D., Milano, M., Silva, A., Thompson, L.: A coalgebraic decision procedure for NetKAT. In: POPL. pp. 343–355 (2015). https://doi.org/10.1145/2676726.2677011
14. Grabmayer, C.: Milner's proof system for regular expressions modulo bisimilarity is complete: Crystallization: Near-collapsing process graph interpretations of regular expressions. In: LICS. pp. 34:1–34:13 (2022). https://doi.org/10.1145/3531130.3532430
15. Grabmayer, C., Fokkink, W.J.: A complete proof system for 1-free regular expressions modulo bisimilarity. In: LICS. pp. 465–478 (2020). https://doi.org/10.1145/3373718.3394744
16. Greenberg, M., Beckett, R., Campbell, E.H.: Kleene algebra modulo theories: a framework for concrete KATs. In: PLDI. pp. 594–608 (2022). https://doi.org/10.1145/3519939.3523722
17. Gumm, H.P.: Functors for coalgebras. Algebra Universalis **45** (11 1998). https://doi.org/10.1007/s00012-001-8156-x
18. Huntington, E.V.: Sets of independent postulates for the algebra of logic. Transactions of the American Mathematical Society **5**(3), 288–309 (1904). https://doi.org/10.1090/S0002-9947-1904-1500675-4
19. Kappé, T., Brunet, P., Rot, J., Silva, A., Wagemaker, J., Zanasi, F.: Kleene algebra with observations. In: CONCUR. pp. 41:1–41:16 (2019). https://doi.org/10.4230/LIPIcs.CONCUR.2019.41
20. Kappé, T., Brunet, P., Silva, A., Wagemaker, J., Zanasi, F.: Concurrent Kleene algebra with observations: From hypotheses to completeness. In: FOSSACS. pp. 381–400 (2020). https://doi.org/10.1007/978-3-030-45231-5_20
21. Kappé, T., Brunet, P., Silva, A., Zanasi, F.: Concurrent Kleene algebra: Free model and completeness. In: ESOP. pp. 856–882 (2018). https://doi.org/10.1007/978-3-319-89884-1_30
22. Kappé, T., Schmid, T., Silva, A.: A complete inference system for skip-free guarded Kleene algebra with tests (2023). https://doi.org/10.48550/arXiv.2301.11301
23. Kleene, S.C.: Representation of events in nerve nets and finite automata. Automata studies **34**, 3–41 (1956)
24. Kot, L., Kozen, D.: Kleene algebra and bytecode verification. Electron. Notes Theor. Comput. Sci. **141**(1), 221–236 (2005). https://doi.org/10.1016/j.entcs.2005.02.028

25. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Inf. Comput. **110**(2), 366–390 (1994). https://doi.org/10.1006/inco.1994.1037
26. Kozen, D.: Kleene algebra with tests and commutativity conditions. In: TACAS. pp. 14–33 (1996). https://doi.org/10.1007/3-540-61042-1_35
27. Kozen, D., Mamouras, K.: Kleene algebra with equations. In: ICALP. pp. 280–292 (2014). https://doi.org/10.1007/978-3-662-43951-7_24
28. Kozen, D., Patron, M.: Certification of compiler optimizations using Kleene algebra with tests. In: CL. pp. 568–582 (2000). https://doi.org/10.1007/3-540-44957-4_38
29. Kozen, D., Smith, F.: Kleene algebra with tests: Completeness and decidability. In: CSL. pp. 244–259 (1996). https://doi.org/10.1007/3-540-63172-0_43
30. Kozen, D., Tseng, W.D.: The Böhm-Jacopini theorem is false, propositionally. In: MPC. pp. 177–192 (2008). https://doi.org/10.1007/978-3-540-70594-9_11
31. Laurence, M.R., Struth, G.: Completeness theorems for pomset languages and concurrent Kleene algebras (2017). https://doi.org/10.48550/arXiv.1705.05896
32. Makowsky, J.A.: Why Horn formulas matter in computer science: Initial structures and generic examples. J. Comput. Syst. Sci. **34**(2/3), 266–292 (1987). https://doi.org/10.1016/0022-0000(87)90027-4
33. Milner, R.: A complete inference system for a class of regular behaviours. J. Comput. Syst. Sci. **28**(3), 439–466 (1984). https://doi.org/10.1016/0022-0000(84)90023-0
34. Pous, D., Rot, J., Wagemaker, J.: On tools for completeness of Kleene algebra with hypotheses. In: RAMICS. pp. 378–395 (2021). https://doi.org/10.1007/978-3-030-88701-8_23
35. Pous, D., Wagemaker, J.: Completeness theorems for Kleene algebra with top. In: CONCUR. pp. 26:1–26:18 (2022). https://doi.org/10.4230/LIPIcs.CONCUR.2022.26
36. Rees, J.: Fizz Buzz: 101 Spoken Numeracy Games. Learning Development Aids (2002)
37. Rutten, J.J.M.M.: Universal coalgebra: a theory of systems. Theor. Comput. Sci. **249**(1), 3–80 (2000). https://doi.org/10.1016/S0304-3975(00)00056-6
38. Salomaa, A.: Two complete axiom systems for the algebra of regular events. J. ACM **13**(1), 158–169 (1966). https://doi.org/10.1145/321312.321326
39. Schmid, T.: A (co)algebraic framework for ordered processes (2022). https://doi.org/10.48550/arXiv.2209.00634
40. Schmid, T., Kappé, T., Kozen, D., Silva, A.: Guarded Kleene algebra with tests: Coequations, coinduction, and completeness. In: ICALP. pp. 142:1–142:14 (2021). https://doi.org/10.4230/LIPIcs.ICALP.2021.142
41. Schmid, T., Rozowski, W., Silva, A., Rot, J.: Processes parametrised by an algebraic theory. In: ICALP. pp. 132:1–132:20 (2022). https://doi.org/10.4230/LIPIcs.ICALP.2022.132
42. Smolka, S., Foster, N., Hsu, J., Kappé, T., Kozen, D., Silva, A.: Guarded Kleene algebra with tests: verification of uninterpreted programs in nearly linear time. In: POPL. pp. 61:1–61:28 (2020). https://doi.org/10.1145/3371129
43. Smolka, S., Kumar, P., Foster, N., Kozen, D., Silva, A.: Cantor meets Scott: semantic foundations for probabilistic networks. In: POPL. pp. 557–571 (2017). https://doi.org/10.1145/3009837.3009843
44. Smolka, S., Kumar, P., Kahn, D.M., Foster, N., Hsu, J., Kozen, D., Silva, A.: Scalable verification of probabilistic networks. In: PLDI. pp. 190–203 (2019). https://doi.org/10.1145/3314221.3314639

45. Takai, T., Furusawa, H.: Monodic tree Kleene algebra. In: RelMICS/AKA. pp. 402–416 (2006). https://doi.org/10.1007/11828563_27
46. Wagemaker, J., Bonsangue, M.M., Kappé, T., Rot, J., Silva, A.: Completeness and incompleteness of synchronous Kleene algebra. In: MPC. pp. 385–413 (2019). https://doi.org/10.1007/978-3-030-33636-3_14
47. Wagemaker, J., Brunet, P., Docherty, S., Kappé, T., Rot, J., Silva, A.: Partially observable concurrent Kleene algebra. In: CONCUR. pp. 20:1–20:22 (2020). https://doi.org/10.4230/LIPIcs.CONCUR.2020.20
48. Zetzsche, S., Silva, A., Sammartino, M.: Guarded Kleene algebra with tests: Automata learning (2022). https://doi.org/10.48550/arXiv.2204.14153

Quorum Tree Abstractions of Consensus Protocols

Berk Cirisci[1](✉), Constantin Enea[2], and Suha Orhun Mutluergil[3]

[1] IRIF, Université Paris Cité, Paris, France
cirisci@irif.fr

[2] LIX, Ecole Polytechnique, CNRS and Institut Polytechnique de Paris, Palaiseau, France
cenea@lix.polytechnique.fr

[3] Sabanci University, Istanbul, Turkey
suha.mutluergil@sabanciuniv.edu

Abstract. Distributed algorithms solving agreement problems like consensus or state machine replication are essential components of modern fault-tolerant distributed services. They are also notoriously hard to understand and reason about. Their complexity stems from the different assumptions on the environment they operate with, i.e., process or network link failures, Byzantine failures etc. In this paper, we propose a novel abstract representation of the dynamics of such protocols which focuses on quorums of responses (votes) to a request (proposal) that form during a run of the protocol. We show that focusing on such quorums, a run of a protocol can be viewed as working over a tree structure where different branches represent different possible outcomes of the protocol, the goal being to stabilize on the choice of a fixed branch. This abstraction resembles the description of recent protocols used in Blockchain infrastructures, e.g., the protocol supporting Bitcoin or Hotstuff. We show that this abstraction supports reasoning about the safety of various algorithms, e.g., Paxos, PBFT, Raft, and HotStuff, in a uniform way. In general, it provides a novel induction based argument for proving that such protocols are safe.

1 Introduction

Consensus or state-machine replication protocols are essential ingredients for maintaining strong consistency in modern fault-tolerant distributed systems. Such protocols must execute in the presence of concurrent and asynchronous message exchanges as well as benign (message loss, process crash) or Byzantine failures (message corruption). Developing practical implementations or reasoning about their correctness is notoriously difficult. Standard examples include the classic Paxos [21] or PBFT [5] protocols, or the more recent HotStuff [37] protocol used in Blockchain infrastructures.

In this paper, we propose a new abstraction for representing the executions of such protocols that can be used in particular, to reason about their safety, i.e., ensuring *Agreement* (e.g., all correct processes decide on a single value) and *Validity* (e.g., the decided value has been proposed by some node participating in the protocol). Usually, protocol executions are composed of a number of communication-closed rounds [11], and each round consists of several phases in which a process broadcasts a request and expects to collect responses from a quorum of processes before advancing to the next phase. The abstraction is defined as a *sequential* object called *Quorum Tree (QTree)* which maintains a tree structure where each node corresponds to a different round in an execution. The operations of QTree, to add or change the status of a node, model quorums of responses that have been received in certain phases of a round.

For instance, a round in single-decree Paxos consists of two phases: a *prepare* phase where a pre-determined leader broadcasts a request for joining that round and expects a quorum of responses from the other processes before advancing to a *vote* phase where it broadcasts a value to agree upon and expects a quorum of responses (votes) in order to declare that value as decided in that round. Rounds are initiated by their respective leaders and can run concurrently. The idea behind QTree is to represent a Paxos execution using a rooted tree where each node different from the root corresponds to a round where the leader has received *a quorum of responses in the prepare phase*. The parent-child relation models the data flow from one round to a later round: responses to join requests contain values voted for in previous rounds (if any) and one of them will be included by the leader in the vote phase request. The round in which that value was voted defines the parent. Then, each node has one out of three possible statuses: `ADDED` if the vote phase can still be successful (the leader can collect a quorum of votes) but this did not happen yet, `GHOST` if the vote phase can not be successful (e.g., a majority of processes advanced to the next round without voting), and `COMMITTED` if the leader has received a quorum of responses in the vote phase. This is a tree structure because before reaching a quorum in the vote phase of a round, other rounds can start and their respective leaders can send other vote requests (with possibly different values). The specific construction of requests and responses in Paxos ensures that all the `COMMITTED` nodes in this tree belong to a *single* branch, which entails the agreement property (this will become clearer when presenting the precise definition of QTree in Section 2).

The QTree abstraction is applicable to a wide range of protocols beyond the single-decree Paxos sketched above. It applies to state-machine replication protocols like Raft [36] and HotStuff [37] where the tree structure represents logs of commands (inputted by clients) stored at different processes and organized according to common prefixes (each node corresponds to a single command) and multi-decree consensus protocols like multi-Paxos [21] and its variants [16,26,23,18], or PBFT [5] where different consensus instances (for different indices in a sequence of commands) are modeled using different QTree instances.

We show that all these protocols are *refinements* of QTree in the sense that their executions can be mapped to sequences of operations on a QTree state, which are about agreeing on a branch of the tree called the *trunk*. These operations are defined as invocations of two methods *add* and *commit* for adding a new leaf to the tree (during which some other nodes may turn to `GHOST`) and changing the status of a node from `ADDED` to `COMMITTED`, respectively. Any sequence of invocations to these methods ensures that all the `COMMITTED` nodes lie on the same branch of the tree (the trunk). In relation to protocol executions, *add* and *commit* invocations that concern the same node correspond to receiving a quorum of responses in two specific phases of a round, which vary from one protocol to another.

The mapping between protocol executions and QTree executions is defined as in proofs of linearizability for concurrent objects with *fixed* linearization points. Analogous to linearizability, where the goal is to show that an object method takes effect instantaneously at a point in time called linearization point, we show that it is possible to mark certain steps of a given protocol as linearization points of *add* or *commit* operations[4], such that the sequence of *add* and *commit* invocations defined by the order between linearization points along a protocol execution is a correct QTree execution. We introduce a declarative characterization of correct QTree executions that simplifies the proof of the latter (see Section 3).

The QTree abstraction offers a novel view on the dynamics of classic consensus or state-machine replication protocols like Paxos, Raft, and PBFT, which relates to the description of recent Blockchain protocols like HotStuff and Bitcoin [27], i.e., agreeing on a branch in a tree. It provides a formal framework to reason *uniformly* about single-decree consensus protocols and state-machine replication protocols like Raft and HotStuff. For single-decree protocols (or compositions thereof), the parent-child relation between QTree nodes corresponds to the data-flow between a quorum of responses to a leader and the request he sends in the next phase while for Raft and HotStuff, it corresponds to an order set by a leader between different commands.

Our work relies on a hypothesis that correctness proofs based on establishing a refinement towards an *operational* specification such as QTree, which can be understood as a sequence of steps, are much more intuitive and "explainable" compared to classic proofs based on inductive invariants. An inductive invariant has to describe all intermediate states produced by all possible orders of receiving messages and a precise formalization is quite complex. As an indication, the Paxos invariant used in recent work [29] (see formulas (4) to (12) in Section 5.2) is a conjunction of eight quantified first-order formulas which are hard to reason about and not re-usable in the context of a different protocol.

We believe that operational specifications are also helpful in taming complexity while designing new protocols or implementations theoreof, or in gaining confidence about their correctness without going through ad-hoc and brittle proof arguments. For instance, our proofs are very clear about the phases of a round in which quorums need to intersect, which provides flexibility and opti-

[4] These linearization points are *fixed* in the sense that they correspond to specific instructions in the code of the protocol, and they do not depend on the future of an execution. For an expert reader, this actually corresponds to a proof of strong linearizability [15].

mization opportunities for deciding on quorum sizes in each phase. Depending on environment assumptions, quorum sizes can be optimized while preserving correctness. Compared to previous operational specifications for reasoning about consensus protocols, e.g., [3,12], QTree is designed to be less abstract so that the refinement proof, establishing the relationship between a given protocol and QTree, is less complex (see Section 8 for details).

2 Quorum Tree

We describe the QTree sequential object which operates on a tree and has two methods *add* and *commit* for adding a new node and modifying an attribute of a node (committing a node), respectively. When used as an abstraction of consensus protocols, invocations of these two methods correspond to certain quorums that are reached during a round of the protocol.

2.1 Overview

QTree is a sequential rooted-tree, a possible state being depicted in Figure 1. The nodes with black dashed margins are not members of the tree and they are discussed later. Each node in the tree contains a round number, a value, and a status field set to `ADDED`, `GHOST`, or `COMMITTED`. The round number acts as an identifier of a node since there can not exist two nodes with the same round number. The *Root* node is part of the initial state and its status is `COMMITTED`. A QTree state consists of a trunk, alive branches, and dead branches; a branch is a chain of nodes connected by the parent relation. Alive branches are extensible with new `ADDED` nodes but dead branches are not. The trunk is a particular branch of the tree that starts from the root. It contains all the `COMMITTED` nodes and it ends with a `COMMITTED` node. It may also contain `ADDED` or `GHOST` nodes. For example, in Figure 1, the trunk consists of *Root* and n_3. All alive branches are connected to the last `COMMITTED` node of the trunk (alive branches can include `ADDED` or `GHOST` nodes). For instance, in Figure 1, the subtree rooted at n_3 contains a single alive branch whose leaf node is n_5. Dead branches can contain only `GHOST` nodes. In Figure 1, the tree contains a single dead branch containing the node n_1.

Nodes can be added to the tree as leaves. The status of a newly added node is either `ADDED` or `GHOST`. The status `ADDED` may turn to `GHOST` or `COMMITTED`. The `GHOST` status is "final" meaning that it can never turn into `COMMITTED` afterwards. However, `GHOST` nodes can be part of alive branches, and they can help in growing the tree.

QTree has two methods *add* and *commit*:

– *add* generates a new leaf with a round number r value v and parent p identified by the round number r_p given as an input. Its status is set to `ADDED` or `GHOST` provided that some conditions hold. If the status of the new node is set as `ADDED`, then it either extends (has a path to the end of) an existing alive branch or creates a new alive branch from the trunk. The new node

may also "invalidate" some other nodes by changing their status from `ADDED` to `GHOST`.
– *commit* extends the trunk by turning the status of a node from `ADDED` to `COMMITTED`. This extension of the trunk may prevent some branches to be extended in the future (some alive branches may become dead), i.e., future invocations of *add* that extend those branches will add only `GHOST` nodes.

Each node models the evolution of a round in a consensus protocol and the value attribute represents the value proposed by the leader of that round. The round and value attributes of a node are immutable and cannot be changed later. We assume that round numbers are strictly positive except for *Root* whose round number is 0.

QTree applies uniformly to a range of consensus or state-machine replication protocols. We start by describing a variation that applies to single-decree consensus protocols, where a number of processes aim to agree on a single value. Multi-decree consensus protocols that are used to solve state-machine replication can be simulated using a number of instances of QTree, one for each decree (the instances are independent one from another). Then, state-machine replication protocols like HotStuff that rely directly on a tree structure to order commands can be simulated by the QTree for single-decree consensus modulo a small change that we discuss later.

2.2 Definition of the Single-Decree Version

Algorithm 1 lists a description of QTree in pseudo-code. The following set of predicates are used as conditions inside methods:

1. $\mathit{link}(n) \equiv n.\text{parent} \in \text{Nodes} \wedge n.\text{parent}.\text{round} < n.\text{round}$
2. $\mathit{newRound}(n) \equiv \forall n' \in \text{Nodes}.\ n'.\text{round} \neq n.\text{round}$
3. $\mathit{maxCommitted}(n) \equiv n.\text{status} = \texttt{COMMITTED} \wedge$ $(\forall n' \in \text{Nodes}.\ n'.\text{status} = \texttt{COMMITTED} \implies n'.\text{round} < n.\text{round})$
4. $\mathit{extendsTrunk}(n) \equiv \exists n' \in \text{Nodes}.\ \mathit{maxCommitted}(n') \wedge$ $(n \text{ extends } n' \vee n.\text{round} < n'.\text{round})$
5. $\mathit{valid}(n) \equiv \mathit{link}(n) \wedge \mathit{newRound}(n) \wedge \mathit{extendsTrunk}(n)$
6. $\mathit{valueConstraint}(n) \equiv n.\text{parent} \neq \mathit{Root} \implies n.\text{value} = n.\text{parent}.\text{value}$

The *add* method (lines 5-17) generates a new node n with round, value, and parent set according to the method's inputs. Then, it adds n to the tree by linking it to the selected parent if n satisfies the following *validity* conditions:

– n's parent belongs to the tree and its round number is smaller than r (predicate *link* at (1)),
– the tree does not contain a node with round number r (predicate *newRound* at (2)),
– if r is bigger than the round number of the last node of the trunk, then n must extend the trunk (predicate *extendsTrunk* at (4)),
– n's value must be the same as its parent's value unless the parent is the *Root* (predicate *valueConstraint* at (6)).

Algorithm 1: The QTree object

```
1  Initialize:
       /* ⊥ denotes non-initialized values */
2      $Root$.round = $\mathbf{0}$; $Root$.status = COMMITTED;
3      $Root$.value = $\perp$; $Root$.parent = $Root$;
4      Nodes = $\{Root\}$;
5  Method add $(r, v, r_p)$
6      Pre: $r > 0$
7      $n$ = new Node(round = $r$, status = $\perp$,
         value = $v$, parent = $p : p.round = r_p$);
8      if $valid(n) \wedge valueConstraint(n)$
9          Nodes = Nodes $\cup \{n\}$;
10         $n$.status = ADDED;
11         if $\exists n' \in Nodes.\ n'.round > n.round$
12             $n$.status = GHOST;
13         forall $n' \in Nodes.\ n'.round < n.round$
14             if $n$ is conflicting with $n'$
15                 $n'$.status ← GHOST;
16         return OK
17     return FAIL
18 Method commit $(r)$
19     if $\exists\ n \in Nodes.\ n.round = r \wedge$
         $n.status$ = ADDED
20         $n$.status ← COMMITTED;
21         return OK
22     return FAIL
```

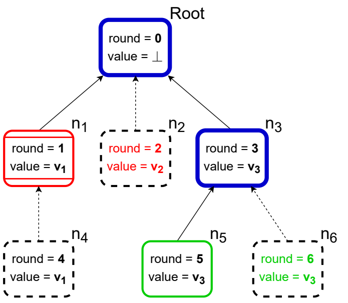


Fig. 1: A state of QTree. We represent ADDED nodes with green solid margins, GHOST nodes with red double-line margins, and COMMITTED nodes with blue thick margins. The nodes with black dashed margins are not part of the state, they are fictitious nodes used to explain the method for adding new nodes.

The *valid* predicate at (5) is the conjunction of the first three constraints.

For example, let us consider an invocation of *add* in a state of QTree that contains the non-dashed nodes in Figure 1. If the invocation generates n_2, n_4, or n_6 (receiving as input the corresponding attributes), then n_2 and n_6 do satisfy all these constraints and can be added to the tree. The node n_4 fails the *extendsTrunk* predicate because it is not extending the last node of the trunk (n_3) and its round number is higher.

If a node n satisfies the conditions above, the *add* method turns its status to either ADDED or GHOST. If there is another node in the tree with a higher round number, n's status becomes GHOST. Otherwise, it becomes ADDED. As a continuation of the example above, the status of n_2 is set to GHOST because the tree contains node n_3 with a higher round number and the status of n_6 is set to ADDED.

Moreover, the addition of n can "invalidate" some other nodes, turn their status to GHOST. This is based on a notion of *conflicting* nodes. We say that two nodes are conflicting if they are on different branches, i.e., there is no path from one node to the other. An *add* invocation that adds a node n changes the

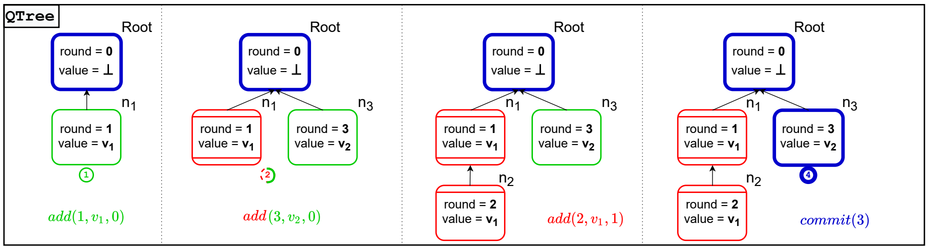


Fig. 2: Explaining the behavior of *add* and *commit* methods. Colors are interpreted as in Fig 1.

status of all the nodes n' in the tree that conflict with n and have a lower round number than n, to GHOST. For example, Figure 2 pictures a sequence of QTree states in an execution, to be read from left to right. The first state represents the result of executing $add(1, v_1, 0)$ on the initial state of QTree, adding node n_1. Executing $add(3, v_2, 0)$ on this first state creates another node n_3 and sets its status to ADDED. This invocation will also turn the status of n_1 to GHOST since its round number is less than the round number of n_3 and they are on different branches. Afterwards, by executing $add(2, v_1, 1)$, a node n_2 is added to the tree with status GHOST since there is a node n_3 on a different branch which has a higher round number.

The method *add* returns OK when the created node is effectively added to the tree (it satisfies the conditions described above) and $FAIL$, otherwise.

Lastly, the *commit* method takes a round number r as input and turns the status of the node containing r to COMMITTED if it was ADDED. If successful, it returns OK and $FAIL$, otherwise. As a continuation of the example above, the right part of Figure 2 pictures a state obtained by executing $commit(3)$ on the state to the left. This sets the status of n_3 to COMMITTED as n_3 was previously ADDED. Note that the conditions in *add* ensure that the tree can not contain two nodes with the same round number.

Safety Properties. We show that the QTree object in Algorithm 1 can be used to reason about the safety of single-decree consensus protocols, in the sense that it satisfies a notion of *Validity* (processes agree on one of the proposed values) and *Agreement* (processes decide on a single value). More precisely, we show that every state that is reachable by executing a sequence of invocations of *add* and *commit* (in Algorithm 1), called simply *reachable state*, satisfies the following:

– *Validity*: every node different from *Root* contains the same value as a child of *Root*, and
– *Agreement*: every two COMMITTED nodes different from *Root* contain the same value.

Proposition 1 (Validity). *Every node in a reachable state that is different from Root contains the same value as a child of Root.*

Proof. A node n is added to the tree only if the predicate *valueConstraint* holds, which implies that it is either a child of *Root* or it has the same value as its parent which is a descendant of *Root*. Also, since the value attribute of a node is immutable, any COMMITTED node contains the same value that it had when it was created by an *add* invocation.

Therefore, the fact that a consensus protocol refining QTree satisfies validity, i.e., processes decide on a value proposed by a client of the protocol, reduces to proving that the phases of a round simulated by *add* invocations that add children of *Root* use values proposed by a client. This is ensured using additional mechanisms, i.e., a client broadcasts its value to all participants in the protocol, so that each participant can check the validity of a value proposed by a leader.

Next, we focus on *Agreement*, and show that COMMITTED nodes belong to a single branch of the tree.

Proposition 2. *Let n_1 and n_2 be two COMMITTED nodes in a reachable state. Then, n_1 and n_2 are not conflicting.*

Proof. Assume towards contradiction that QTree reaches a state where two COMMITTED nodes n_1 and n_2 are conflicting. Let $r_1 = n_1$.round and $r_2 = n_2$.round. Without loss of generality, we assume that $r_1 < r_2$. Such a state is reachable if $add(r_1, \_, \_)$ and $add(r_2, \_, \_)$ resulted in adding the nodes n_1 and n_2 and set their status to ADDED (we use _ to denote arbitrary values), and subsequently, $commit(r_1)$ and $commit(r_2)$ switched the status of both n_1 and n_2 to COMMITTED. If $add(r_1, \_, \_)$ were to execute before $add(r_2, \_, \_)$, then $add(r_2, \_, \_)$ would have changed the status of n_1 to GHOST because it is conflicting with n_2. Otherwise, if $add(r_2, \_, \_)$ were to execute before $add(r_1, \_, \_)$, then the latter would have set the status of n_1 to GHOST since the tree contains n_2 that has a higher round number. In both cases, executing $commit(r_1)$ can never turn the status of n_1 to COMMITTED.

Proposition 2 allows to conclude that any two COMMITTED nodes (different from *Root*) contain the same value. Indeed, a node can become COMMITTED only if it was ADDED, which implies that is has the same value as its parent (the predicate *valueConstraint* holds), and by transitivity, as any of its ancestors, except for *Root*.

Proposition 3 (Agreement). *Let n_1 and n_2 be two COMMITTED nodes in a reachable state, which are different from Root. Then, $n_1.value = n_2.value$.*

2.3 State Machine Replication Versions

The single-decree version described above can be extended easily to a *multi-decree* context. As multi-decree consensus protocols, used in state machine replication, can be seen as a composition of multiple instances of single-decree consensus protocols, a multi-decree version of QTree is obtained by composing multiple instances of the single-decree version. Each of these instances manipulates a tree as described above without interference from other instances. The validity and agreement properties above apply separately to each instance.

The single-decree version can also be extended for state machine replication protocols like HotStuff and Raft where the commands (values) are a-priori structured as a tree, i.e., each command given as input is associated to a predetermined parent in this tree. Then, the goal of such a protocol is to agree on a sequence in which to execute these commands, i.e., a branch in this tree. Simply removing the *valueConstraint* condition in the *add* method (underlined in Algorithm 1) enables QTree to simulate such protocols. A node's value need not be the same as its parent's value to be valid for *add*. Proposition 2 that implies the agreement property of such protocols still holds (Proposition 3 does not hold when the *valueConstraint* condition is removed; this property is specific to single-decree consensus). Since the value field remains immutable, the validity property of such protocols reduces to ensuring that the values generated during phases simulated by *add* correspond to commands issued by the client (Proposition 1 is also specific to single-decree consensus and it does not hold). As before, this requires additional mechanisms, i.e., a client broadcasting a command to all the participants in the protocol, whose correctness can be established quite easily.

3 Consensus Protocols Refining QTree

In the following, we show that a number of consensus protocols are refinements of QTree in the sense that their executions can be mimicked with *add* and *commit* invocations. This is similar to a linearizable concurrent object being mimicked with invocations of a sequential specification. The refinement relation allows to conclude that the *Validity* and *Agreement* properties of QTree imply similar properties for any of its refinements.

The definition of the refinement relation relies on a formalization of protocols and QTree as *labeled transition systems*. For a given protocol, a state is a tuple of process local states and a set of messages in transit, and a transition corresponds to an indivisible step of a process (receiving a set of messages, performing a local computation step, or sending a message). For QTree, a state is a tree of nodes as described above and a step corresponds to an invocation to *add* or *commit*. An execution is a sequence of transitions from the initial state.

Refinement corresponds to a mapping between protocol executions and QTree executions. This mapping is defined as in proofs of linearizability for concurrent objects with *fixed linearization points*, where the goal is to show that each concurrent object method appears to take effect instantaneously at a point in time that corresponds to executing a fixed statement in its code. Therefore, certain steps of a given protocol are considered as linearization points of *add* and *commit* QTree invocations (returning OK), and one needs to prove that the sequence of invocations defined by the order of linearization points in a protocol execution is a correct execution of QTree.

Formally, a labeled transition system (LTS) is a tuple $L = (\mathcal{Q}, q_0, \mathcal{T}, \mathcal{A}_L)$ where $\mathcal{Q}$ is a set of states, q_0 is the unique initial state, $\mathcal{A}_L$ is a set of actions (transition labels) and $\mathcal{T}$ is a set of transitions (q, a, q') such that $q, q' \in \mathcal{Q}$ and $a \in \mathcal{A}_L$. An *execution* E from q_0 is a finite sequence of alternating states and actions such that $E = q_0, a_0, q_1, a_1, \ldots, q_n$ with $(q_i, a_i, q_{i+1}) \in \mathcal{T}$ for each

$0 \leq i \leq n-1$. A trace t is the sequence of actions projected from some execution E. $T(L)$ denotes the set of traces of L.

The standard notion of refinement between LTSs states that an LTS L is a refinement of another LTS L' when $T(L) \subseteq T(L')$. In this paper, we consider a slight variation of this definition of refinement that applies to LTSs that do *not* share the same set of actions, representing for instance, some concrete protocol and QTree, respectively. This notion of refinement is parametrized by a mapping Γ between actions of L and L', respectively. We say that L *Γ-refines* L' when $\Gamma(T(L)) \subseteq T(L')$. Here, a mapping $\Gamma : \mathcal{A}_L \rightarrow \mathcal{A}_{L'}$ is extended to sequences and sets of sequences as expected, e.g., $\Gamma(a_1 \ldots a_n) = \Gamma(a_1) \ldots \Gamma(a_n)$. With this extension, the preservation of safety specifications from an LTS to a refinement of it requires certain constraints on the mapping Γ that will be discussed in Section 4.2.

In the context of proving that a concrete protocol refines QTree, the goal is to define a mapping Γ between actions of the protocol and QTree *add*/*commit* invocations such that Γ applied to protocol executions results in correct QTree executions. In the following, we provide a characterization of correct QTree executions that simplifies such refinement proofs.

3.1 Characterizing QTree Invocation Sequences

An *invocation label* $add(r, v, r_p) \Rightarrow RET$ or $commit(r) \Rightarrow RET$ combines a QTree method name with input values and a return value $RET \in \{OK, FAIL\}$. An invocation label is called *successful* when the return value is OK. A sequence σ of invocation labels is called *correct* when there exist QTree states $q_0, \ldots, q_{|\sigma|}$, such that q_0 is the QTree initial state and for each $i \in [1, |\sigma|]$, executing σ_i starting from q_{i-1} leads to q_i.

Theorem 1. *A sequence σ of successful invocation labels is correct if and only if the following hold (we use _ to denote arbitrary values):*

1. *for every r, σ contains at most one invocation label $add(r, \_, \_)$ and at most one invocation label $commit(r)$*
2. *every $commit(r)$ is preceded by an $add(r, \_, \_)$*
3. *if $r_p > 0$, every $add(r, v, r_p)$ is preceded by $add(r_p, v', \_)$ where $0 < r_p < r$ (a) and $v = v'$*
4. *if σ contains $add(r, \_, \_)$ and $add(r', \_, r'')$ with $r'' < r < r'$, then σ does* not *contain $commit(r)$*

Properties 1–3 are straightforward consequences of the *add* and *commit* definitions. Indeed, it is impossible to add two nodes with the same round number r, which implies that there can not be two successful $add(r, \_, \_)$ invocations, the status of a node can be flipped to `COMMITTED` exactly once, which implies that there can not be two successful $commit(r)$ invocations, and a $commit(r)$ is successful only if a node with round number r already exists, hence Property 2 must hold. Moreover, a node's parent defined by the input r_p must already exist in the tree, which implies that Property 3 must also hold. Property 4 is more involved and relies on the fact that a node n with round number r can be `COMMITTED` only if there exist no other conflicting node n' with a bigger round number r' (the parent of n' having a round smaller than r implies that n and n' are conflicting).

Proof. ($\Rightarrow$): Assume that σ is correct. We show that it satisfies the above properties:

- Property 1: The $newRound(n)$ predicate used at line 8 in Algorithm 1 ensures that it is impossible to add two nodes with the same round number r, and therefore σ can not contain two successful $add(r, \_, \_) \Rightarrow OK$ invocations. The conditions at line 19 ensure that $commit(r) \Rightarrow OK$ can flip the status of a node only once, and therefore only one such successful invocation can occur in σ.
- Property 2: The conditions at line 19 in Algorithm 1 imply that the state in which $commit(r) \Rightarrow OK$ is executed contains a node with round number r. This node could have only added by a previous $add(r, \_, \_) \Rightarrow OK$ invocation.
- Property 3: The $link(n)$ predicate used at line 8 in Algorithm 1 ensures that the state in which $add(r, v, r_p) \Rightarrow OK$ is executed contains a node with round number r_p. This node could have only added by a previous $add(r_p, v', \_) \Rightarrow OK$ invocation, for some v'.
  - Property 3a: It is a direct consequence of the $valueConstraint(n)$ predicate used at line 8 in Algorithm 1.
- Property 4: Let n and n' be the nodes of the QTree state q reached after executing σ, which have been added by $add(r, \_, \_) \Rightarrow OK$ and $add(r', \_, r'') \Rightarrow OK$, respectively. We have that n'.round $>$ n.round $>$ n.parent.round. Since the round numbers decrease when going from one node towards *Root* in a reachable QTree state, it must be the case that n and n' are conflicting. By Lemma 1, we get that n.status is `GHOST`. Since the `GHOST` status can not be turned to `COMMITTED` and vice-versa, it follows that σ can not contain $commit(r) \Rightarrow OK$.

($\Leftarrow$): We prove that every sequence σ that satisfies properties 1–4 is correct. We proceed by induction on the size of σ. The base step is trivial. For the induction step, let σ be a sequence of size $k + 1$. If σ satisfies properties 1-4, then the prefix σ' containing the first k labels of σ satisfies properties 1-4 as well. By the induction hypothesis, σ' is correct. We show that the last invocation of σ, denoted by σ_{k+1} can be executed in the QTree state $q_{|\sigma'|}$ reached after executing σ'. We start with a lemma stating an inductive invariant for reachable QTree states:

Lemma 1. *For every node n in any state q reached after executing a correct sequence σ of successful invocations, n.status is* `COMMITTED` *if n is Root or σ contains a $commit(r)$ invocation. Else, n.status is* `GHOST` *if q contains a node n' with n'.round $>$ n.round and n' is conflicting with n, and it is* `ADDED`*, otherwise.*

Proof. We proceed by induction on the size of σ. The base step is trivial. For the induction step, let σ be a sequence of size $m+1$. Let q_m be the state reached after executing the prefix of size m of σ, and let σ_{m+1} be the last invocation label of σ. We show that the property holds for any possible σ_{m+1} that takes the QTree state q_m to some other state q_{m+1}:

- $\sigma_{m+1} = add(r, v, r_p) \Rightarrow OK$, for some r, v, r_p: Let n be the new node added by this invocation. The status of n can be `ADDED` or `GHOST`. If q_m contains a node n' with n'.round $> r$ (since round numbers are decreasing going towards the *Root* and n is a new leaf node, any existing node with a higher round number such as n' is also conflicting with n), then the status of n becomes `GHOST` by the predicate at line 11 in Algorithm 1 (otherwise, it remains `ADDED`). This implies that n's status satisfies the statement in the lemma. This invocation may also turn the status of some set of nodes N from `ADDED` to `GHOST` by the statement at line 13 in Algorithm 1. The nodes in N have a lower round number than r and conflicting with n. Therefore, the statement of the lemma is satisfied for the nodes in N.
- $\sigma_{m+1} = commit(r) \Rightarrow OK$, for some r: For $commit(r)$ to be successful the conditions at line 19 in Algorithm 1 must be satisfied. If it is satisfied, only the status of node n is changed from `ADDED` to `COMMITTED`. Note that *Root* exists by definition and its status is `COMMITTED`. Since the statuses of the rest of the nodes stay the same, the statement of the lemma holds. □

There are two cases to consider depending on whether σ_{k+1} is an *add* or *commit* invocation label:

- $add(r, v, r_p)$: This invocation label is successful if and only if the predicates $valid(n)$ and $valueConstraint(n)$ at line 8 in Algorithm 1 are satisfied after generating a new node n with the given inputs in the state $q_{|\sigma'|}$:
  - $newRound(n)$: Due to Property 1, $r \neq n'$.round for any other node $n' \in q_{|\sigma'|}$ and the predicate is satisfied.
  - $link(n)$: To satisfy this predicate, there must exist a node in $q_{|\sigma'|}$ with round r_p where $r_p < r$. By Property 3, if σ contains $add(r, \_, r_p) \Rightarrow OK$ with $r_p \neq 0$, then $add(r_p, \_, \_) \Rightarrow OK$ also exists in σ. Hence, there exists a node p with round r_p in $q_{|\sigma'|}$, and the predicate is satisfied. If $r_p = 0$, then $q_{|\sigma'|}$ contains the *Root* node (with round 0) which ensures that the predicate is satisfied.
  - $extendsTrunk(n)$: This predicate states that n extends the node n' which has the highest round number among the nodes with `COMMITTED` status, if n.round $> n'$.round. Assume by contradiction that this is not the case, i.e., n.round $> n'$.round but n and n' are conflicting. Let n_1 be the lowest common ancestor of n and n' (the first common node on the paths from n and n' to the *Root*). Since the round numbers decrease when going from one node towards *Root*, we have that n_1.round $< n'$.round. If we consider the nodes on the path from n to n_1, since n.round $>$ n'.round, there must exist a node n_2 such that n_2.round $> n'$.round but n_2.parent.round $< n'$.round. The node n_2 in $q_{|\sigma'|}$ corresponds to the

invocation label $add(n_2.\text{round}, \_, n_2.\text{parent.round})$ in σ'. Moreover, the `COMMITTED` status of n' implies the existence of $commit(n'.\text{round})$ in σ' as stated in Lemma 1. However, it is impossible that σ' contains both these invocation labels if Property 4 holds.

  - $valueConstraint(n)$: It is implied trivially as Property 3a holds.

– $commit(r)$: It is successful if and only if the conditions at line 19 in Algorithm 1 are satisfied. Then by Property 1 and 2, there exist $add(r, \_, \_)$ in σ' but not $commit(r)$. As $add(r, \_, \_)$ is successful, there already exist a node n in $q_{|\sigma'|}$ where its round is r but its status can be either `ADDED` or `GHOST`. Towards a contradiction, assume that $n.\text{status} = $ `GHOST` in $q_{|\sigma'|}$. This means that there exists a node n' conflicting with n such that $n'.\text{round} > n.\text{round}$ as stated in Lemma 1. Let n_1 be the least common ancestor of n and n'. Since round numbers are decreasing going towards the $Root$, $n_1.\text{round} < n.\text{round}$. If we consider nodes on the path from n' to n_1, there exists a node n_2 such that $n_2.\text{round} > n.\text{round}$ and $n_2.\text{parent.round} < n.\text{round}$. That's why, there is an invocation label $add(n_2.round, \_, n_2.parent.round)$ in σ'. However, σ cannot contain both of these invocation labels together according to Property 4. □

4 Linearization Points

We describe an instrumentation of consensus protocols with linearization points of successful QTree invocations, and illustrate it using Paxos as a running example. Section 5 and Section 6 will discuss other protocols like HotStuff, Raft, PBFT, and multi-Paxos. This instrumentation defines the mapping Γ between actions of a protocol and QTree, respectively, such that the protocol is a Γ-refinement of QTree. We also discuss the properties of this instrumentation which imply that establishing Γ-refinement is an effective proof for the safety of the protocol.

The identification of linearization points relies on the fact that protocol executions pass through a number of rounds, and each round goes through several phases (rounds can run asynchronously – processes need not be in the same round at the same time). The protocol imposes a total order over the phases inside a round and among distinct rounds. Processes executing the protocol can only move forward following the total order on phases/rounds. Going from one phase to the next phase in the same round is possible if a quorum of processes send a particular type of message. The refinement proofs require identifying two quorums for each round where a value is first proposed to be agreed upon and then decided. They correspond to linearization points of successful $add(r, \_, \_)$ and $commit(r)$, respectively. The linearization point of $add(r, v, r_p) \Rightarrow OK$ occurs when intuitively, the value v is proposed as a value to agree upon in round r. For the protocols we consider, v is determined by a designated leader after receiving a set of messages from a quorum of processes. For single-decree consensus, members of the quorum send the latest round number and value they adopted (voted) in the past and the leader picks a value corresponding to the maximum round number r_p. If no one in the quorum has adopted any value yet, then the leader is free to propose any value received from a client, and r_p equals a default value 0. For state-machine replication protocols like HotStuff or Raft, the round r_p is defined in a different manner – see Section 5 (and the full version of this work [9]). The linearization point of $commit(r) \Rightarrow OK$ occurs when a quorum of nodes adopt (vote for) a value v proposed at round r.

By Theorem 1, proving that the order between linearization points along a protocol execution defines a correct QTree execution reduces to showing Properties 1–4. In general, Properties 1–3 are quite straightforward to establish and follow from the control-flow of a process. Property 3a is specific to single-decree consensus protocols or compositions thereof, e.g., (multi-)Paxos and PBFT. It will not hold for Raft or Hotstuff. Property 4 is related to the fact that any two quorums of processes intersect in a correct process.

Above, we have considered the case of a protocol that is a refinement of a single instance of QTree. State machine replication protocols that are composed of multiple independent consensus instances, e.g., PBFT (see Section 6), are refinements of a set of QTree instances (identified using a sequence number) and every linearization point needs to be associated with a certain QTree instance.

4.1 Linearization Points for Paxos

For concreteness, we exemplify the instrumentation with linearization points on the single-decree Paxos protocol. We start with a brief description of this protocol that focuses on details relevant to this instrumentation.

Paxos proceeds in rounds and each round has a unique leader. Since the set of processes running the protocol is fixed and known by every process, the leader of each round can be determined by an a-priorly fixed deterministic procedure (e.g., the leader is defined as $r \ mod \ N$ where r is the round number and N the number of processes). For each round, the leader acts as a proposer of a value to agree upon.

A round contains two phases. In the first phase, the leader broadcasts a `START` message to all the processes to start the round, executing the **`START`** action below, and processes acknowledge with a `JOIN` message if some conditions are met, executing the **`JOIN`** action:

- **`START` Action:** The leader p of round $r > 0$ (the proposer) broadcasts a `START`(r) message to all processes.
- **`JOIN` Action:** When a process p' receives a `START`(r) message, if p' has not sent a `JOIN` or `VOTE` message (explained below) for a higher round in the past[5], it replies by sending a `JOIN`(r) message to the proposer. This message includes the maximum round number ($maxVotedRound$) for which p' has sent a `VOTE` message in the past and the value ($maxVotedValue$) proposed in that round. If it has not voted yet, these fields are 0 and $\perp$.

[5] Each process has a local variable $maxJoinedRound$ that stores the maximal round it has joined or voted for in the past and checks whether $maxJoinedRound < r$

If the leader receives `JOIN` messages from a quorum of processes, i.e., at least $f+1$ processes from a total number of $2f+1$, the second phase starts. The leader broadcasts a `PROPOSE` message with a value, executing the **`PROPOSE`** action below. Processes may acknowledge with a `VOTE` message if some conditions are met, executing a **`VOTE`** action. If the leader receives `VOTE` messages from a quorum of processes, then the proposed value becomes decided (and sent to the client) by executing a **`DECIDE`** action:

- **`PROPOSE` Action:** When the proposer p receives `JOIN`(r) messages from a quorum of $(f+1)$ processes, it selects the one with the highest vote round number and proposes its value by broadcasting a `PROPOSE`(r) message (which includes that value). If there is no such highest round (all vote rounds are 0), then the proposer selects the proposed value randomly simulating a value given by the client (whose modeling we omit for simplicity).
- **`VOTE` Action:** When a process p' receives a `PROPOSE`(r) message, if p' has not sent a `JOIN` or `VOTE` message for a higher round in the past, it replies by sending a `VOTE`(r) message to the proposer with round number r.
- **`DECIDE` Action:** When the proposer p receives `VOTE`(r) messages from a quorum of processes, it updates a local variable called *decidedVal* to be the value it has proposed in this round r. This assignment means that the value is decided and sent to the client.

Linearization points in Paxos. We instrument Paxos with linearization points as follows:

- the linearization point of $\mathit{add}(r, v, r') \Rightarrow \mathit{OK}$ occurs when the proposer broadcasts the `PROPOSE`(r) message containing value v after receiving a quorum of `JOIN`(r) messages (during the **`PROPOSE`** action in round r). The round r' is extracted from the `JOIN`(r) message selected by the proposer.
- the linearization point of $\mathit{commit}(r) \Rightarrow \mathit{OK}$ occurs when the leader of round r updates *decidedVal* after receiving a quorum of `VOTE`(r) messages (during the **`DECIDE`** Action).

We illustrate the definition of linearization points for Paxos in relation to QTree executions in the full version [9].

Theorem 2. *Paxos refines QTree.*

Proof. We show that the sequence of successful *add* and *commit* invocations defined by linearization points along a Paxos execution satisfies the properties in Theorem 1 and therefore, it represents a correct QTree execution:

- **Property 1:** Each round has a unique leader and the leader follows the rules of the protocol (no Byzantine failures), thereby, making a single proposal. Therefore, the linearization point of an $\mathit{add}(r, \_, \_) \Rightarrow \mathit{OK}$ will occur at most once for a round r. Since a single value can be proposed in a round, and all processes follow the rules of the protocol, they can only vote for that single value. Thus, at most one linearization point of $\mathit{commit}(r) \Rightarrow \mathit{OK}$ can occur for a round r.

- **Property 2:** This holds trivially as all the processes follow the rules of the protocol and they need to receive a PROPOSE(r) message (which can occur only after the linearization point of an $add(r, \_, \_) \Rightarrow OK$) from the leader of round r to send a VOTE(r) message.
- **Property 3:** By the definition of the **PROPOSE** action, the proposer selects a highest vote round number r' from a quorum of JOIN(r) messages that it receives, before broadcasting a PROPOSE(r) message. If such a highest vote round number $r' > 0$ exists, then there must be a VOTE(r') message which is a reply to a PROPOSE(r') message. Thus, if the linearization point of $add(r, \_, r') \Rightarrow OK$ occurs where $r' \neq 0$, then it is preceded by $add(r', \_, \_)$. Also, by the definition of **JOIN**, a process can not send a JOIN(r) message after a VOTE(r') message if $r \not> r'$.
  - **Property 3a:** By the definition of **PROPOSE**, the proposer selects the JOIN message with the highest vote round number and proposes its value. Thus, if the linearization points of both $add(r, v, r') \Rightarrow OK$ and $add(r', v', \_) \Rightarrow OK$ occur, then $v = v'$.
- **Property 4:** Assume by contradiction that the linearization point of $commit(r) \Rightarrow OK$ occurs along with the linearization points of $add(r, \_, \_) \Rightarrow OK$ and $add(r', \_, r'') \Rightarrow OK$, for some $r'' < r < r'$. The linearization point of $commit(r)$ occurs because of a quorum of VOTE(r) messages sent by a set of processes P_1, and $add(r', \_, r'')$ occurs because of a quorum of JOIN(r') messages sent by a set of processes P_2. Since P_1 and P_2 must have a non-empty intersection, by the definition of **JOIN**, it must be the case that $r'' \geq r$, which contradicts the hypothesis.

The proof of Property 4 relies exclusively on the quorum of processes in the first phase of a round intersecting the quorum of processes in the second phase of a round. It is not needed that quorums in first, resp., second, phases of different rounds intersect. This observation is at the basis of an optimization that applies to non-Byzantine protocols like Flexible Paxos [18] or Raft (see the full version [9]).

4.2 Inferring Safety

The main idea behind these linearization points is that successful *add* and *commit* invocations correspond to some process doing a step that witnesses for the receipt a quorum of messages sent in a certain phase of a round. Intuitively, linearization points of successful *add* invocation occur when some process in some round is certain that a quorum of processes received or will receive the same proposal (same value, parent etc.) for the same round and acts accordingly (sends a message). Such proposal on a value v in a round r is denoted by the linearization point of successful $add(r, v, r')$ for some r'. On the other hand, the linearization point of a successful $commit(r)$ invocation occurs when a process decides on a value in round r (e.g., after receiving a quorum of votes). Formally, if we denote the actions of a protocol that correspond to linearization points of successful $add(r, v, r')$ and $commit(r)$ invocations using a_a and a_c, respectively, then $\Gamma(a_a) = add(r, v, r') \Rightarrow OK$ and $\Gamma(a_c) = commit(r) \Rightarrow OK$.

When the protocol is such a Γ-refinement of QTree, then, it satisfies agreement and validity. If a decision on a value v in a round r of a protocol is the linearization point of a successful $commit(r)$, then by Theorem 1, the corresponding QTree state contains a node n with n.round $= r$, n.value $= v$, and n.status $=$ `COMMITTED`. For single-decree consensus, Proposition 3 ensures that all rounds decide on the same value. For state machine replication protocols like Raft and HotStuff, where the goal is to agree on a sequence of commands, Proposition 2 ensures that all the decided values lie on the same branch of the tree which ensures that all processes agree on the same sequence of commands.

For validity, when $valueConstraint(n)$ is considered, successful $add(r, v, 0)$ invocations represent proposals of client values. Theorem 1 ensures that these invocations correspond to nodes n that are immediate children of *Root* and for any such node n, n.value $= v$. Therefore, by Proposition 1, we can conclude that only client values can be decided. When $valueConstraint(n)$ is not considered, the fact that the value of each node is obtained from a client is ensured using additional mechanisms that are straightforward, e.g., a client broadcasting a command to all the participants in the protocol.

5 HotStuff Refines QTree

We present an instrumentation of HotStuff with linearization points of successful *add* and *commit* invocations. We use HotStuff as an example of a state machine replication protocol where processes agree over a sequence of commands to execute, and any new command proposed by a leader to the other processes comes with a well-identified immediate predecessor in this sequence. Agreement over a command entails agreement over all its predecessors in the sequence. This is different from protocols such as multi-Paxos or PBFT, discussed in the next section, where commands are associated to indices in the sequence and they can be agreed upon in any order. Instrumentation of Raft is presented in the full version [9] and behaves in a similar manner.

In HotStuff, f out of a total of $N = 3f + 1$ processes might be Byzantine in the sense that they might show arbitrary behavior and send corrupt or spurious messages. However, they are limited by cryptographic protocols. HotStuff requires that messages are signed using public-key cryptography, which implies that Byzantine processes cannot imitate messages of correct (non-faulty) processes. Additionally, after receiving a quorum of messages, leaders must include certificates in their own messages to prove that a quorum has been reached. These certificates are constructed using threshold signature schemes and correct processes will not accept any message from the leader if it is not certified. Because of Byzantine processes, HotStuff requires quorums of size of $2f + 1$ which ensures that the intersection of any two quorums contains at least one correct process.

Each process stores a tree of commands. When a node in this tree (representing some command) is decided, all the ancestors of this node in the tree (nodes on the same branch) are also decided. For a node to become decided, a leader must receive a quorum of messages in 3 consecutive phases after the proposal. After each quorum is established, the leader broadcasts a different certificate to state which quorum has been achieved and the processes update different local variables accordingly, with the same node (if the certificate is valid). These local variables are $preNode$, $votedNode$ and $decidedNode$ in the order of quorums.

To start a new round, processes send their $preNode$'s to the leader of the next round in `ROUND-CHANGE(r)` messages and increment their round number. After getting a quorum of messages and selecting the $preNode$ with the highest round, the leader broadcasts a `PROPOSE(r)` message with a new node (value is taken from the client) whose parent is the selected $preNode$. When the message is received by a process, it first checks if the new node extends the selected $preNode$. Then it accepts the new node if the node extends its own $votedNode$ (it is a descendant of $votedNode$ in the tree) or it has a higher round number than the round number of its $votedNode$, and sends[6] a `JOIN(r)` message with the same content. In the second (resp., third) phase, if a quorum of `JOIN(r)` (resp., `PRECOMMIT_VOTE(r)`) messages is received by the leader, it broadcasts a `PRE-COMMIT(r)` (resp., `COMMIT(r)`) message, and processes update their $preNode$ (resp., $votedNode$) with the new node, sending a `PRECOMMIT_VOTE(r)` (resp., `COMMIT_VOTE(r)`) message. In the fourth phase, when the leader receives a quorum of `COMMIT_VOTE(r)`, it broadcasts a `DECIDE(r)` message and processes update their $decidedNode$ accordingly. See the full version [9] for more details.

For HotStuff, the linearization points of *add* and *commit* occur with the broadcasts of `PRECOMMIT`(r) and `DECIDE`(r) messages, respectively, that are *valid* , i.e., (1) they contain certificates for quorums of `JOIN(r)` or `COM-MIT_VOTE(r)` messages, respectively, which respect the threshold signature scheme, and (2) they contain the same node as in those messages. More precisely,

- the linearization point of $add(r, v, r') \Rightarrow OK$ occurs the *first* time when a *valid* `PRECOMMIT`(r) message containing node v is sent. r' is the round of the node which is the parent of v and it is contained in a previous `PROPOSE(r)` message (r' can be 0 in which case parent of v is a distinguished root node that exists in the initial state).
- the linearization point of $commit(r) \Rightarrow OK$ occurs the *first* time when a *valid* `DECIDE`(r) message is sent.

Note that a Byzantine leader can send multiple *valid* `PRECOMMIT`(r) messages that include certificates for different quorums of `JOIN(r)` messages. A linearization point occurs when the first such message is sent. Even if processes reply to another valid `PRECOMMIT`(r) message sent later, this later `PRECOMMIT`(r) message contains the same $preNode$ value, and their reply will have the same content. The same holds for `DECIDE`(r) messages. This remark along with the restriction

[6] For all received messages, a correct process also checks if the round number of the node sent by the leader is equal to the current round number of its own, and can send only one message for each phase in each round.

to valid messages and the fact that any two quorums intersect in at least one correct process implies that the sequence of successful *add* and *commit* invocations defined by these linearization points satisfies the properties in Theorem 1 and therefore,

Theorem 3. *HotStuff refines QTree.*

A detailed proof of the theorem above is given in the full version [9].

6 PBFT Refines QTree

The protocols discussed above are refinements of a *single* instance of QTree. State-machine replication protocols based Multi-decree consensus like Multi-Paxos or PBFT can be seen as compositions of a number of single-decree consensus instances that run concurrently, one for each index in a sequence of commands to agree upon, and they are refinements of a set of independent QTree instances. We describe the instrumentation of PBFT and delegate multi-Paxos (and variants) to the full version [9].

PBFT is a multi-decree consensus protocol in which processes aim to agree on a sequence of values. As in HotStuff, f out of a total number of $3f + 1$ processes might be Byzantine and quorums are of size at least $2f + 1$. To ensure authentication, messages are signed using public-key cryptography. Messages sent after receiving a quorum of messages in a previous phase include that set of messages as a certificate.

A new round r starts with the leader receiving a quorum of `ROUND-CHANGE`(r) messages (like in HotStuff). Each such message from a process p includes the `VOTE` message with the highest round (similarly to the **`JOIN`** action of Paxos) that p sent in the past, for each sequence number that is not yet agreed by a quorum. For an arbitrary set of sequence numbers sn, the leader selects the `VOTE` message with the highest round and broadcasts a `PROPOSE`(r,sn) message that includes the same value as in the `VOTE` message or a value received from a client if there is no such highest round. As mentioned above, this message also includes the `VOTE` messages that the leader received as a certificate for the selection. When a process receives a `PROPOSE`(r,sn) message, if r equals its current round, the process did not already acknowledge a `PROPOSE`(r,sn) message, and the value proposed in this message is selected correctly w.r.t. the certificate, then it broadcasts a `JOIN`(r,sn) message with the same content (this is sent to all processes not just the leader). If a quorum of `JOIN`(r,sn) messages is received by a process, then it broadcasts a `VOTE`(r,sn) message with the same content. If a process receives a quorum of `VOTE`(r,sn) messages, then the value in this message is decided for sn. When a process sends its highest round number `VOTE` messages to the leader of the next round (in `ROUND-CHANGE` messages), it also includes the quorum of `JOIN` messages that it received before sending the `VOTE`, as a certificate.

PBFT is a refinement of a set of independent QTree instances, one instance for each sequence number. The linearization points will refer to a specific instance identified using a sequence number, e.g., $sn.add(r, v, r')$ denotes an $add(r, v, r')$ invocation on the QTree instance sn. Therefore,

- the linearization point of $sn.add(r, v, r') \Rightarrow OK$ occurs the *first* time when a process p sends a `VOTE`$(r,\ sn)$ message, assuming that p is *honest*, i.e., it already received a quorum of `JOIN`$(r,\ sn)$ messages with the same content. v is the value of the `VOTE`$(r',\ sn)$ message that is included in the `PROPOSE`(r,sn) message (it is possible that $r' = 0$ and v is selected randomly).
- the linearization point of $sn.commit(r) \Rightarrow OK$ occurs the *first* time when a process p decides a value for sn, assuming that p is *honest*, i.e., it already received a quorum of `JOIN`$(r,\ sn)$, resp., `VOTE`$(r,\ sn)$, messages with the same content.

A protocol refines a set of QTree instances identified using sequence numbers when it satisfies Properties 1-4 in Theorem 1 for each sequence number, e.g., Property 1 becomes for every sn and every r, a protocol execution contains a linearization point for at most one invocation $sn.add(r, \_, \_) \Rightarrow OK$ and at most one invocation $sn.commit(r) \Rightarrow OK$. A detailed proof of the following theorem is given in the full version [9].

Theorem 4. *PBFT refines a composition of independent QTree instances.*

7 Discussion

Protocols considered in this work can be grouped under three classes: single-decree consensus (Paxos), multi-decree consensus (PBFT, Multi-Paxos) and state machine replication (Raft, HotStuff)[7]. We show that they all refine QTree: a single instance for Paxos and HotStuff, and a set of independent instances (one for each sequence number in a command log) for PBFT, Multi-Paxos, and Raft. The more creative parts of the refinement proofs are the identification of *add* and *commit* linearization points and establishing Property 4 in Theorem 1 which follows from the intersection of quorums achieved in different phases of a round. The other 3 properties in Theorem 1 which guarantee that the linearization points are correct are established in a rather straightforward manner, based on the control-flow of a process participating to the protocol.

The linearization points of successful *add* and *commit* invocations correspond to some process doing a step that witnesses for the receipt a quorum of messages sent in a certain phase of a round, e.g., the leader broadcasting a `PROPOSE`(r) message in Paxos entails that a quorum of `JOIN`(r) messages have been sent in the first phase and received. Protocols vary in the total number of phases in a round, and the phases for which quorums of sent messages should be received in order to have a linearization point of *add* or *commit*. A summary is presented in Table 1. The * on the total number of phases means that the first phase is skipped in rounds where the leader is stable. For Multi-Paxos and Raft, if the first phase

[7] This is a slight abuse of terminology since multi-decree consensus protocols are typically used to implement state machine replication.

is skipped, then the linearization point of an *add* is determined by a quorum of received messages sent in the next phase (and coincides with the linearization point of a *commit*). We use "1/2" to denote this fact. In PBFT and HotStuff, due to Byzantine processes, quorums of messages sent in two consecutive phases need to be received in order to ensure that the processes are going to vote on the same valid proposal. The 3rd phase in HotStuff is used to ensure progress and can be omitted when reasoning only about safety.

Table 1: Summary of linearization point definitions. For each protocol, we give the total number of phases in a round and the number of the phase for which a quorum of sent messages should be received in order to have a linearization point of *add* or *commit*.

Class	Protocol	#Phases	*add* Quorum Pha.	*commit* Quorum Pha.
Single-Decree Cons.	Paxos	2	1	2
Multi-Decree Cons.	Multi-Paxos	2*	1/2	2
	PBFT	3*	2	3
State Machine Repl.	Raft	2*	1/2	2
	HotStuff	4	2	4

8 Conclusion and Related Work

We have proposed a new methodology for proving safety of consensus or state-machine replication protocols, which relies on a novel abstraction of their dynamics. This abstraction is defined as a sequential QTree object whose state represents a global view of a protocol execution. The operations of QTree construct a tree structure and model agreement on values or a sequence of state-machine commands as agreement on a fixed branch in the tree. Our methodology applies uniformly to a range of protocols like (multi-)Paxos, HotStuff, Raft, and PBFT. We believe that this abstraction helps in improving the understanding of such protocols and writing correct implementations or optimizations thereof.

As a limitation, it is not clear whether QTree applies to protocols such as Texel [31] which do not admit a decomposition in rounds. As future work, we might explore the use of QTree in reasoning about liveness. This would require some fairness condition on infinite sequences of add/commit invocations, and a suitable notion of refinement which ensures that infinite sequences of protocol steps cannot be mapped to infinite sequences of stuttering QTree steps.

The problem of proving the correctness of such protocols has been studied in previous work. We give an overview of the existing approaches that starts with safety proof methods based on refinement, which are closer to our approach.

Refinement based safety proofs. Verdi [35] is a framework for implementing and verifying distributed systems that contains formalizations of various network semantics and failure models. Verdi provides *system transformers* useful for refining high-level specifications to concrete implementations. As a case study, it includes a fully-mechanized correctness proof of Raft [36]. This proof consists of 45000 lines of proof code (manual annotations) in the Coq language for a 5000 lines RAFT implementation, showing the difficulty of reasoning on consensus protocols and the manual effort required. Iron Fleet [17] uses TLA [22] style transition-system specifications and refine them to low-level implementations described in the Dafny programming language [25]. Boichat et al. [3] defines a class of specifications for consensus protocols, which are more abstract than QTree and can make correctness proofs harder. Proving Paxos in their case is reduced to a linearizability proof towards an abstract specification, which is quite complex because the linearization points are *not fixed*, they depend on the future of an execution. As a possibly superficial quantitative measure, their Paxos proof reduces to 7 lemmas that are formalized by Garcia-Perez et al. [12,13] in 12 pages (see Appendix B and C in [13]), much more than our QTree proof. Our refinement proof is also similar to a linearizability proof, but the linearization points in our case are *fixed* (do not depend on the future of an execution) which brings more simplicity. In principle, the specifications in [3] could apply to more protocols, but we are not aware of such a case. The inductive sequentialization proof rule [20] is used for a fully mechanized correctness proof of a realistic Paxos implementation. This implementation is proved to be a refinement of a *sequential* program which is quite close to the original implementation, much less abstract than QTree, and relies on commutativity arguments implied by the communication-closed round structure [11]. A similar idea is explored in [14], but in a more restricted context.

Inductive invariant based safety proofs. Ivy [30] is an SMT-based safety verification tool that can be used for verifying inductive invariants about global states of a distributed protocol. In order to stay in a decidable fragment of first-order logic, both the modeling and the specification language of IVY are restricted. A simple model of Paxos obeying these restrictions is proven correct in [29].

Beyond safety. The TLA+ infrastructure [22] of Lamport has been used to verify both safety and liveness (termination) of several variations of Paxos, e.g., Fast Paxos [23] or Multi-Paxos [6]. Bravo et al. [4] introduce a generic synchronization mechanism for round changes, called the view synchronizer, which guarantees liveness for various Byzantine consensus protocols including our cases studies HotStuff and PBFT. This work includes full correctness proofs for single-decree versions of HotStuff and PBFT and a two-phase version of HotStuff. PSync [10] provides a partially synchronous semantics for distributed protocols assuming communication-closed rounds in the Heard-Of model [8]. PSync is used to prove both safety and liveness of a Paxos-like consensus protocol called *lastVoting*.

Relating different consensus protocols. Lamport defines a series of refinements of Paxos that leads to a Byzantine fault tolerant version, which is refined by PBFT [24]. Our proof that Paxos refines QTree can be easily extended to this Byzantine fault tolerant version in the same manner as we did for PBFT.

Wang et al. [34] shows that a variation of RAFT is a refinement of Paxos, which enables porting some Paxos optimizations to RAFT. Renesse et al. [32] compare Paxos, Viewstamped Replication [28] and ZAB [19]. They define a rooted tree of specifications represented in TLA style whose leaves are concrete protocols. Each node in this tree is refined by its children. Common ancestors of concrete protocols show similarities whereas conflicting specifications show the differences. Similarly, [33] shows that Paxos, Chandra-Toueg [7] and Ben-Or [2] consensus algorithms share common building blocks. Aublin et al. [1] propose an abstract data type for specifying existing and possible future consensus protocols. Unlike our QTree, core components of this data type are not implemented and intentionally left abstract so that it can adapt to different network and process failure models.

References

1. Aublin, P., Guerraoui, R., Knezevic, N., Quéma, V., Vukolic, M.: The next 700 BFT protocols. ACM Trans. Comput. Syst. **32**(4), 12:1–12:45 (2015). https://doi.org/10.1145/2658994, https://doi.org/10.1145/2658994
2. Ben-Or, M.: Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In: Probert, R.L., Lynch, N.A., Santoro, N. (eds.) Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983. pp. 27–30. ACM (1983). https://doi.org/10.1145/800221.806707, https://doi.org/10.1145/800221.806707
3. Boichat, R., Dutta, P., Frølund, S., Guerraoui, R.: Deconstructing paxos. SIGACT News **34**(1), 47–67 (2003). https://doi.org/10.1145/637437.637447, https://doi.org/10.1145/637437.637447
4. Bravo, M., Chockler, G.V., Gotsman, A.: Making byzantine consensus live. In: Attiya, H. (ed.) 34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference. LIPIcs, vol. 179, pp. 23:1–23:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.DISC.2020.23, https://doi.org/10.4230/LIPIcs.DISC.2020.23
5. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Seltzer, M.I., Leach, P.J. (eds.) Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999. pp. 173–186. USENIX Association (1999), https://dl.acm.org/citation.cfm?id=296824
6. Chand, S., Liu, Y.A., Stoller, S.D.: Formal verification of multi-paxos for distributed consensus. In: Fitzgerald, J.S., Heitmeyer, C.L., Gnesi, S., Philippou, A. (eds.) FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9995, pp. 119–136 (2016). https://doi.org/10.1007/978-3-319-48989-6_8, https://doi.org/10.1007/978-3-319-48989-6_8
7. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996). https://doi.org/10.1145/226643.226647, https://doi.org/10.1145/226643.226647

8. Charron-Bost, B., Merz, S.: Formal verification of a consensus algorithm in the heard-of model. Int. J. Softw. Informatics **3**(2-3), 273–303 (2009), http://www.ijsi.org/ch/reader/view_abstract.aspx?file_no=273&flag=1
9. Cirisci, B., Enea, C., Mutluergil, S.O.: Quorum tree abstractions of consensus protocols (2023). https://doi.org/10.48550/ARXIV.2301.09946, https://arxiv.org/abs/2301.09946
10. Dragoi, C., Henzinger, T.A., Zufferey, D.: Psync: a partially synchronous language for fault-tolerant distributed algorithms. 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. 400–415. ACM (2016). https://doi.org/10.1145/2837614.2837650, https://doi.org/10.1145/2837614.2837650
11. Elrad, T., Francez, N.: Decomposition of distributed programs into communication-closed layers. Sci. Comput. Program. **2**(3), 155–173 (1982). https://doi.org/10.1016/0167-6423(83)90013-8, https://doi.org/10.1016/0167-6423(83)90013-8
12. García-Pérez, Á., Gotsman, A., Meshman, Y., Sergey, I.: Paxos consensus, deconstructed and abstracted. In: Ahmed, A. (ed.) Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10801, pp. 912–939. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_32, https://doi.org/10.1007/978-3-319-89884-1_32
13. García-Pérez, Á., Gotsman, A., Meshman, Y., Sergey, I.: Paxos consensus, deconstructed and abstracted (extended version). CoRR **abs/1802.05969** (2018), http://arxiv.org/abs/1802.05969
14. von Gleissenthall, K., Kici, R.G., Bakst, A., Stefan, D., Jhala, R.: Pretend synchrony: synchronous verification of asynchronous distributed programs. Proc. ACM Program. Lang. **3**(POPL), 59:1–59:30 (2019). https://doi.org/10.1145/3290372, https://doi.org/10.1145/3290372
15. Golab, W.M., Higham, L., Woelfel, P.: Linearizable implementations do not suffice for randomized distributed computation. In: Fortnow, L., Vadhan, S.P. (eds.) Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011. pp. 373–382. ACM (2011). https://doi.org/10.1145/1993636.1993687, https://doi.org/10.1145/1993636.1993687
16. Gray, J., Lamport, L.: Consensus on transaction commit. CoRR **cs.DC/0408036** (2004), http://arxiv.org/abs/cs.DC/0408036
17. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: proving practical distributed systems correct. In: Miller, E.L., Hand, S. (eds.) Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015. pp. 1–17. ACM (2015). https://doi.org/10.1145/2815400.2815428, https://doi.org/10.1145/2815400.2815428
18. Howard, H., Malkhi, D., Spiegelman, A.: Flexible paxos: Quorum intersection revisited. CoRR **abs/1608.06696** (2016), http://arxiv.org/abs/1608.06696
19. Junqueira, F.P., Reed, B.C., Serafini, M.: Zab: High-performance broadcast for primary-backup systems. In: Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011. pp. 245–256. IEEE Compute Society (2011). https://doi.org/10.1109/DSN.2011.5958223, https://doi.org/10.1109/DSN.2011.5958223

20. Kragl, B., Enea, C., Henzinger, T.A., Mutluergil, S.O., Qadeer, S.: Inductive sequentialization of asynchronous programs. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 227–242. ACM (2020). https://doi.org/10.1145/3385412.3385980, https://doi.org/10.1145/3385412.3385980
21. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998). https://doi.org/10.1145/279227.279229, https://doi.org/10.1145/279227.279229
22. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002), http://research.microsoft.com/users/lamport/tla/book.html
23. Lamport, L.: Fast paxos. Distributed Comput. **19**(2), 79–103 (2006). https://doi.org/10.1007/s00446-006-0005-x, https://doi.org/10.1007/s00446-006-0005-x
24. Lamport, L.: Byzantizing paxos by refinement. In: Peleg, D. (ed.) Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6950, pp. 211–224. Springer (2011). https://doi.org/10.1007/978-3-642-24100-0_22, https://doi.org/10.1007/978-3-642-24100-0_22
25. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20, https://doi.org/10.1007/978-3-642-17511-4_20
26. Malkhi, D., Lamport, L., Zhou, L.: Stoppable paxos. Tech. Rep. MSR-TR-2008-192 (April 2008)
27. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep. (2008), https://bitcoin.org/bitcoin.pdf
28. Oki, B.M., Liskov, B.: Viewstamped replication: A general primary copy. In: Dolev, D. (ed.) Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988. pp. 8–17. ACM (1988). https://doi.org/10.1145/62546.62549, https://doi.org/10.1145/62546.62549
29. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: decidable reasoning about distributed protocols. Proc. ACM Program. Lang. **1**(OOPSLA), 108:1–108:31 (2017). https://doi.org/10.1145/3140568, https://doi.org/10.1145/3140568
30. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. 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. 614–630. ACM (2016). https://doi.org/10.1145/2908080.2908118, https://doi.org/10.1145/2908080.2908118
31. van Renesse, R.: Asynchronous consensus without rounds. CoRR **abs/1908.10716** (2019), http://arxiv.org/abs/1908.10716
32. van Renesse, R., Schiper, N., Schneider, F.B.: Vive la différence: Paxos vs. viewstamped replication vs. zab. IEEE Trans. Dependable Secur. Comput. **12**(4), 472–484 (2015). https://doi.org/10.1109/TDSC.2014.2355848, https://doi.org/10.1109/TDSC.2014.2355848

33. Song, Y.J., van Renesse, R., Schneider, F.B., Dolev, D.: The building blocks of consensus. In: Rao, S., Chatterjee, M., Jayanti, P., Murthy, C.S.R., Saha, S.K. (eds.) Distributed Computing and Networking, 9th International Conference, ICDCN 2008, Kolkata, India, January 5-8, 2008. Lecture Notes in Computer Science, vol. 4904, pp. 54–72. Springer (2008). https://doi.org/10.1007/978-3-540-77444-0_5, https://doi.org/10.1007/978-3-540-77444-0_5
34. Wang, Z., Zhao, C., Mu, S., Chen, H., Li, J.: On the parallels between paxos and raft, and how to port optimizations. In: Robinson, P., Ellen, F. (eds.) Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019. pp. 445–454. ACM (2019). https://doi.org/10.1145/3293611.3331595, https://doi.org/10.1145/3293611.3331595
35. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: Grove, D., Blackburn, S.M. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 357–368. ACM (2015). https://doi.org/10.1145/2737924.2737958, https://doi.org/10.1145/2737924.2737958
36. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.E.: Planning for change in a formal verification of the raft consensus protocol. In: Avigad, J., Chlipala, A. (eds.) Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016. pp. 154–165. ACM (2016). https://doi.org/10.1145/2854065.2854081, https://doi.org/10.1145/2854065.2854081
37. Yin, M., Malkhi, D., Reiter, M.K., Golan-Gueta, G., Abraham, I.: Hotstuff: BFT consensus with linearity and responsiveness. In: Robinson, P., Ellen, F. (eds.) Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019. pp. 347–356. ACM (2019). https://doi.org/10.1145/3293611.3331591, https://doi.org/10.1145/3293611.3331591

MAGπ: Types for Failure-Prone Communication

Matthew Alan Le Brun(✉) and Ornela Dardha(✉)

University of Glasgow, Glasgow, UK
m.le-brun.1@research.gla.ac.uk
ornela.dardha@glasgow.ac.uk

Abstract. *Multiparty Session Types* (MPST) are a typing discipline for communication-centric systems, guaranteeing communication safety, deadlock freedom and protocol compliance. Several works have emerged which model failures and introduce fault-tolerance techniques. However, such works often make assumptions on the underlying network, *e.g.*, assuming TCP-based communication where messages are guaranteed to be delivered; or adopting centralised reliable nodes and ad-hoc notions of reliability; or only addressing a single kind of failure, such as node crashes. In this work, we develop MAGπ—a Multiparty, Asynchronous and Generalised π-calculus, which is the *first language and type system* to accommodate in unison: (*i*) the widest range of non-Byzantine faults, including *message loss, delays* and *reordering*; *crash* and *link failures*; and *network partitioning*; (*ii*) a novel and most general notion of *reliability*, taking into account the viewpoint of *each* participant in the protocol; (*iii*) a spectrum of network assumptions from the lowest UDP-based network programming to the TCP-based application level. We prove subject reduction and session fidelity; process properties (deadlock freedom, termination, *etc.*); failure-handling safety and reliability adherence.

Keywords: Session types · Distributed protocols · Failures · Timeouts

1 Introduction

Despite large investments into fault-prevention techniques, failures still regularly occur in complex distributed applications. It is widely agreed that traditional methods of verification using software testing do not provide high levels of confidence in the correctness of distributed algorithms. This is mainly due to the *non-deterministic* behaviour inherent to these protocols, which makes it unfeasible to manually test for all edge cases. This problem is bypassed by using exhaustive techniques such as model checking [9,31], capable of exploring the entirety of the state space of a program to verify its correctness. However, building suitable models for complex distributed algorithms is arduous, expensive, and often intractable (due to the state explosion problem [10]). Furthermore, even if an algorithm is successfully encoded into a suitable model and checked, guarantees of correctness are on the *design* of the algorithm, and not on the software *implementation*; handwritten code is still prone to human error. Contrastively, types

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 363–391, 2023.
https://doi.org/10.1007/978-3-031-30044-8_14

and type systems [29] are lightweight forms of verification. Baked in programming languages, types provide guarantees directly on handwritten code and aid developers in implementing software which is correct by construction. Specific to concurrent and distributed computing, *session types* [14,35,15,36,33,16] have quickly grown in popularity since their initial conceptualisation [14], spanning from *binary*–two participants, to *multiparty*–many participants.

Session types enforce that processes communicate according to a protocol specification. Consequently, desirable properties about communication, *e.g.*, *type safety* (communication occurs error-free), *protocol compliance* (or session fidelity; processes behave according to their predefined protocol), and *deadlock freedom* (processes do not get stuck waiting), can be *statically* determined by a type checker. To this aim, session types have been implemented in various programming languages, including Java [18,11], Go [21], Haskell [17,27], Scala [32], Rust [19], Elixir [34].

To date, most session type theories are designed for *concurrent*, as opposed to *distributed* processes—*i.e.,* it is commonly assumed that communication failures do not occur. For the few (and rapidly increasing) works that do consider failures, heavy assumptions are made that impede their viability for realistic complex distributed applications. *E.g.*, *asynchronous* theories [24,16,33] use *message buffers* to model distributed communication under "TCP-like" assumptions: messages are guaranteed to be delivered and messages from a single sender do not get reordered. *Affine sessions* [25,12,6] only allow failure-handling of application level failures through *try-catch* blocks; there is no support for *arbitrary* failures that may stem from hardware faults, network inconsistencies *etc. Coordinator model* approaches [1,8,37] assume some degree of reliability, be it as a central resilient process, a reliable broadcast, or fixed synchronisation points.

The harsh reality is that many real-world distributed protocols (*e.g.*, consensus algorithms) cannot assume *any* of these conditions. Networks introduce many points of failure into a system: nodes may crash, messages can be dropped, delayed or duplicated, links between nodes may fail *etc.* Designers of distributed protocols have acknowledged that failure is inevitable, and so algorithms are designed to withstand a threshold of failure whilst still achieving their expected behaviour—known as *fault-tolerance* [22]. Examples of fault-tolerant protocols (extensively) used today include the Paxos [20] and Raft [26] consensus algorithms, which assume the possibility of *all non-Byzantine* faults—*i.e.,* node crashes, link failures, network partitions, and message inconsistencies.

Although the correctness of these algorithms has been heavily studied, many of them are developed with limited confidence in the correctness of the deployable artifact, due to the reasons previously outlined. To fill this gap, we need type-based verification, which can be made available to programming languages, thus supporting designers and developers in designing and implementing correct distributed algorithms. While (multiparty) session types have made great impact in modelling structured communication and guaranteeing relevant properties, their theory is not yet expressive to model these complex algorithms.

In this paper, we take steps towards filling this gap by presenting MAGπ—a Multiparty, Asynchronous and Generalised π-calculus—the first language and type system able to accommodate: (*i*) the widest range of non-Byzantine faults, including *message loss*, *delays* and *reordering*; *crash* and *link failures*; and *network partitioning*—all by using *timeouts*; (*ii*) a novel and most general notion of *reliability*, taking into account the viewpoint of each participant in the protocol; and (*iii*) a spectrum of network assumptions—from the lowest level of network programming based on UDP, to application level based on TCP.

Example 1 (Ping Pong: Types). We illustrate MAGπ with a simplified version of the ping utility from the Internet Control Message Protocol (ICMP[1]), which is our running example. The ping utility consists of a total of three *roles* communicating amongst each other: two roles, p and r, communicate *reliably* with each other, and both communicate *unreliably* with a third role q. Our definition of reliability (§ 3.2) takes into account the viewpoint of each role, thus allowing roles to have their own (possibly empty) *reliability set*. Following the assumptions above, the reliability set for p is $\{\mathsf{r}\}$, for r is $\{\mathsf{p}\}$, and for q is $\emptyset$.

Below we give the session types, denoted $\mathbb{S}_{\mathsf{r}}$, $\mathbb{S}_{\mathsf{p}}$ and $\mathbb{S}_{\mathsf{q}}$ for roles r, p and q respectively.

$$\mathbb{S}_{\mathsf{r}} = \&\{\mathsf{p}\,?\,\mathsf{ok}().\mathbf{end},\ \mathsf{p}\,?\,\mathsf{ko}().\mathbf{end}\}$$

$$\mathbb{S}_{\mathsf{p}} = \mathsf{q}\,!\,\mathsf{ping}().\,\&\left\{\begin{array}{l}\mathsf{q}\,?\,\mathsf{pong}().\mathsf{r}\,!\,\mathsf{ok}().\mathbf{end},\\ \text{⏲}.\,\mathsf{q}\,!\,\mathsf{ping}().\,\&\left\{\begin{array}{l}\mathsf{q}\,?\,\mathsf{pong}().\mathsf{r}\,!\,\mathsf{ok}().\mathbf{end},\\ \text{⏲}.\,\mathsf{q}\,!\,\mathsf{ping}().\,\&\left\{\begin{array}{l}\mathsf{q}\,?\,\mathsf{pong}().\mathsf{r}\,!\,\mathsf{ok}().\mathbf{end},\\ \text{⏲}.\,\mathsf{r}\,!\,\mathsf{ko}().\mathbf{end}\end{array}\right.\end{array}\right.\end{array}\right.$$

$$\mathbb{S}_{\mathsf{q}} = \&\left\{\begin{array}{l}\mathsf{p}\,?\,\mathsf{ping}().\mathsf{p}\,!\,\mathsf{pong}().\mathbf{end},\\ \text{⏲}.\ \&\left\{\begin{array}{l}\mathsf{p}\,?\,\mathsf{ping}().\mathsf{p}\,!\,\mathsf{pong}().\mathbf{end},\\ \text{⏲}.\ \&\left\{\begin{array}{l}\mathsf{p}\,?\,\mathsf{ping}().\mathsf{p}\,!\,\mathsf{pong}().\mathbf{end},\\ \text{⏲}.\,\mathbf{end}\end{array}\right.\end{array}\right.\end{array}\right.$$

Role r is the *receiver* (&–called *branching*), which waits on two options: it receives from p either the label ok or ko and then it terminates the protocol (**end**). Role p is the *sender* ($\oplus$[2]–called *selection*), and it tries to obtain information on the status of q. It begins by sending a ping message to q ($\mathsf{q}\,!\,\mathsf{ping}()$), then waits to receive from q. If a pong is received ($\mathsf{q}\,?\,\mathsf{pong}()$) in the top branch, then it concludes that the status of q is *reachable* and sends this information to r ($\mathsf{r}\,!\,\mathsf{ok}()$), after which it terminates. Alternatively, p enters a *timeout branch* (⏲). For simplicity, we assume p will attempt to communicate with q three times (shown in the three-time indentation of the timeout branch) before assuming q is *unreachable*; after which the session will also terminate by sending ko to r, followed by **end**. In the same lines, the protocol for role q is given by the session type $\mathbb{S}_{\mathsf{q}}$, where its timeout branches match the timeouts from $\mathbb{S}_{\mathsf{p}}$.

[1] https://www.rfc-editor.org/rfc/rfc792

[2] For readability, we adopt a shorthand notation for sending towards a single role and for payloads of type unit, such that $\oplus\{\mathsf{s}\,!\,\mathsf{m}(\mathtt{unit}).\mathbb{S}\}$ is represented by $\mathsf{s}\,!\,\mathsf{m}().\mathbb{S}$.

1.1 Contributions

We now present our contributions w.r.t. our Multiparty, Asynchronous, and Generalised π-calculus (MAGπ).

1. **MAGπ** language (§ 2):
   - MAGπ is *the first language* to support the widest set of non-Byzantine faults, including **message loss**, **message delays** and **message reordering**; **crash failures** and **link failures**; and **network partitioning**.
   - MAGπ is *the first language* to introduce **timeouts** in receive branches (used for handling network failures), as well as support **undirected branching** in a *generalised* setting—the ability to simultaneously expect an incoming message from more than one sender.
2. **MAGπ** type system (§ 3):
   - is a *conservative extension* of a generalised asynchronous MPST theory [33], benefiting from: the ability to model *more protocols* than traditional syntactic theories (*e.g.* global types); and the flexibility of checking desired properties, such as deadlock freedom or termination, *a posteriori*—as opposed to during the design phase.
   - supports **undirected branching/selection** and is the first type system to introduce **timeout branches**.
   - supports a novel and most general *reliability* definition (§ 3.2), taking into account the viewpoint of *each* participating role, and is built on *optional role-dependant* reliability assumptions.
3. **Type properties** (§ 4): We prove subject reduction (theorem 1) and session fidelity (theorem 2). We show failure-handling safety (cor. 1) and its inverse, reliability adherence (cor. 2), which strictly connect timeouts and reliability. We prove process properties (theorem 3) *e.g.* deadlock-freedom, termination, liveness, in line with [33]. Finally, as our MAGπ type system is Turing-complete, we prove decidable type checking (theorem 4) and decidability of process properties for finite message buffers (theorem 5).
4. **TCP vs. UDP** (§ 5): MAGπ is expressive enough to capture a range of network assumptions: from low-level network programming operating over the User Datagram Protocol (UDP); to application-level software operating over the Transmission Control Protocol (TCP).
5. **Case study** (§ 6): we further demonstrate the use of timeouts and undirected branching to model a Domain Name System (DNS) distributed protocol with a cache and in-built load-balancer; we also show the properties it satisfies, including safety and deadlock freedom. Further examples are available in the technical report [23], including a prototype specification of a leader election algorithm used by consensus protocols.

2 MAGπ: Language

We present a multiparty session π-calculus, based on the theory of Scalas and Yoshida [33], extended to accurately model real-world distributed network environments. We assume the *lowest level* of abstraction—the only *failure detection mechanism* available to a process is an upper-bound wait limit, *i.e.,* a *timeout.*

$$
\begin{array}{rlr}
c ::= & x \mid s[\mathsf{p}] & (\textit{variable, session w/ role}) \\
d ::= & v \mid c & (\textit{basic value, variable, session w/ role}) \\
w ::= & v \mid s[\mathsf{p}] & (\textit{basic value, session w/ role}) \\
P, Q ::= & \mathbf{0} \mid (\nu s)\, P & (\textit{inaction, restriction}) \\
& \mid P \,|\, Q \mid P + Q & (\textit{composition, non-deterministic choice}) \\
& \mid c \oplus [\mathsf{q}]\,!\,\mathsf{m}\langle d\rangle .\, P & (\textit{selection towards role } \mathsf{q}) \\
& \mid c\, \&_{i\in I}\{[\mathsf{q}_i]\,?\,\mathsf{m}_i(x_i).P_i\} & (\textit{reliable branching from roles } \mathsf{q}_i) \\
& \mid c\, \&_{i\in I}\{[\mathsf{q}_i]\,?\,\mathsf{m}_i(x_i).P_i,\ \ \circlearrowright .\, Q\} & (\textit{branching from roles } \mathsf{q}_i \textit{ w/ } \textbf{\textit{timeout}}\ Q) \\
& \mid \mathsf{def}\ D\ \mathsf{in}\ P \mid X\langle \tilde{d}\rangle & (\textit{process definition, process call}) \\
& \mid s : \sigma & (\textit{session buffer}) \\
D ::= & X(\tilde{x}) = P & (\textit{process declaration}) \\
\sigma ::= & (\mathsf{p} \triangleright \mathsf{q} \triangleleft \mathsf{m}\langle w\rangle) \cdot \sigma \mid \epsilon & (\textit{session queue, empty session queue})
\end{array}
$$

Fig. 1. Syntax for MAGπ

Our calculus presents three novel features: *(i)* the new *timeout* primitive; *(ii)* the capability of expecting a message from *different senders*; and *(iii)* operational semantics which can model *various non-Byzantine failures*. Timeouts can be attached to receive actions—henceforth referred to as *branches*—and are used to describe an alternative process to be executed in case failures are *assumed* to occur (akin to error handlers).

Failures are said to be *assumed*, as opposed to detected, since we model the impossibility result of distinguishing between a delayed *vs* lost message. Thus, it is possible for a processes to prematurely timeout without its corresponding message having been lost—just like the real-world!

The benefit of our approach is that the *failure detection mechanism is agnostic to the type of fault*, allowing us to model in unison **message loss**, **message delay**, **crash-stop failures**, **link failures**, and **network partitions**.

2.1 Syntax

Definition 1 (Language Syntax). *The Multiparty, Asynchronous and Generalised π-calculus syntax is defined by the grammar in fig. 1.*

Communication happens over sessions (s, s') between a number of roles (p, q) ranging over set $\boldsymbol{\rho}$. The primitives of the calculus are *sessions with roles* $s[\mathsf{p}]$, and basic values v, both of which can be abstracted using *variables* (x, y). Processes (P, Q), include the following standard constructs: *(i) inaction* $\mathbf{0}$ represents process termination; *(ii)* session *restriction* $(\nu s)\, P$ binds a new session s in P; *(iii)* parallel *composition* declares two concurrent processes; *(iv) selection* $c\oplus[\mathsf{q}]\,!\,\mathsf{m}\langle d\rangle .\, P$ uses channel c to send a message to q with label m and payload

$$[\textbf{R-}\oplus]\quad s[\mathsf{p}]\oplus[\mathsf{q}]\,!\,\mathsf{m}\langle w\rangle.\,P \;\mid\; s{:}\sigma \longrightarrow P \;\mid\; s{:}\sigma\cdot(\mathsf{p}\triangleright\mathsf{q}\triangleleft\mathsf{m}\langle w\rangle)\cdot\epsilon$$

$$[\textbf{R-}\&]\quad s[\mathsf{q}]\,\&_{i\in I}\{[\mathsf{p}_i]\,?\,\mathsf{m}_i(x_i).P_i\ [,\ \circledcirc.\,Q]\} \;\mid\; s{:}(\mathsf{p}_k\triangleright\mathsf{q}\triangleleft\mathsf{m}_k\langle w\rangle)\cdot\sigma \longrightarrow P_k[^{w}/_{x_k}] \;\mid\; s{:}\sigma \qquad \text{for } k\in I$$

$$[\textbf{R-}\circledcirc]\quad s[\mathsf{q}]\,\&_{i\in I}\{[\mathsf{p}_i]\,?\,\mathsf{m}_i(x_i).P_i,\ \circledcirc.\,Q\} \;\mid\; s{:}\sigma \longrightarrow Q \;\mid\; s{:}\sigma$$

$$[\textbf{R-+}]\quad P_1{+}P_2 \longrightarrow P_i \qquad \text{for } i\in\{1,2\}$$

$$[\textbf{R-}X]\quad \mathsf{def}\ X(x_1,\ldots,x_n)=P\ \mathsf{in}\ (X\langle w_1,\ldots,w_n\rangle \mid Q) \longrightarrow \mathsf{def}\ X(x_1,\ldots,x_n)=P\ \mathsf{in}\ (P[^{w_1}/_{x_1}]\cdots[^{w_n}/_{x_n}] \mid Q)$$

$$[\textbf{R-}\mathbb{C}]\quad P\longrightarrow P' \implies \mathbb{C}[P]\longrightarrow\mathbb{C}[P']$$

$$[\textbf{R-}\downarrow]\quad s{:}h\cdot\sigma\longrightarrow s{:}\sigma$$

Fig. 2. Reduction rules for MAGπ

d—after sending, the process continues according to P; *(v) definition* and *declaration* allow processes to be assigned names, modelling recursion through the use of process *calls*. We now elaborate on the novelties in our language.

$\mathbf{c}\,\&_{i\in I}\{[\mathsf{q}_i]\,?\,\mathbf{m}_i(d).P\}$ Quantification over roles in a branch allows processes to receive from one in a range of other roles. This has practical applications in a multitude of distributed protocols, *e.g. load balancers.*

$\mathbf{c}\,\&_{i\in I}\{[\mathsf{q}_i]\,?\,\mathbf{m}_i(d).P,\ \circledcirc.\,Q\}$ Timeouts are used as a *failure detection mechanism* in receive branches. If a failure is assumed to have lost or prevented the incoming message, then process Q is initiated. It is key to note that timeouts are non-deterministic—they model an arbitrary and unknown duration of time a process waits before assuming a failure has occurred.

$P+Q$ Non-deterministic choice randomly picks between two possible process continuations. We use this construct to simplify examples for better presentation. Concretely, it replaces the need for *expressions* and *if-then-else* constructs, which are routine and orthogonal to our formulation.

$s:\sigma$ Message buffer for session s. An entry in the buffer $(\mathsf{p}\triangleright\mathsf{q}\triangleleft\mathsf{m}\langle w\rangle)$ models a message "in transit" from role p to q with label m and payload w. This is needed to accommodate asynchrony in our language.

2.2 Operational Semantics

We begin with definitions of a reduction context and buffer congruence.

Definition 2 (Reduction Context). *A* ***reduction context*** $\mathbb{C}$ *abstracts away an outer environment from a process, and is given by:*

$$\mathbb{C}\ ::=\ \mathbb{C}\,|\,P \;\mid\; (\nu s)\,\mathbb{C} \;\mid\; \mathsf{def}\ D\ \mathsf{in}\ \mathbb{C} \;\mid\; [\,]$$

Hence, $\mathbb{C}[P]$ *refers to process* P *under some arbitrary context* $\mathbb{C}$.

Definition 3 (Buffer Congruence). *A process containing **only** a buffer under its restriction is congruent to inaction. Message buffers observe total reordering.*

$$(\nu s)\, s : \sigma \;\equiv\; \mathbf{0} \qquad\qquad s : \sigma_1 \cdot h_1 \cdot h_2 \cdot \sigma_2 \;\equiv\; s : \sigma_1 \cdot h_2 \cdot h_1 \cdot \sigma_2$$

Definition 4 (OS). *The operational semantics for MAGπ is given via a* reduction relation $\longrightarrow$ *inductively defined in fig. 2, together with standard* structural congruence rules *[33] and two* buffer congruence rules *defined in def. 3.*

Let us now comment on the reduction rules (fig. 2). Processes send messages using the **selection rule** [**R-**$\oplus$]; this adds the sent message as a new entry in the session buffer, and advances the process to its continuation. Sent messages are read from the buffer using the **branching rule** [**R-** &]. If the receiver has a valid branch matching the sender and message label, then it advances to the specific continuation of said branch (a timeout branch for this rule is optional). The substitution $P_k[{}^{w}/_{x_k}]$ denotes the replacement of variable x_k with the payload value w in the continuation process P_k. The **timeout rule** [**R-**⏲] advances processes to their timeout branch *without* changing the buffer. Non-deterministic choice is resolved using the **choice rule** [**R-**+], which advances the process to one of the two possible continuations. The **call rule** [**R-*X***] replaces a process call with its defined process, substituting each parameter. Processes can reduce under a context using the **context rule** [**R-**$\mathbb{C}$]. Lastly, messages can be lost from the buffer with the **drop rule** [**R-**$\downarrow$].

We now unpack how our semantics deals with failures. The reduction rules in fig. 2 allow various forms of failures to be modelled, stemming from the versatility and elegance of the drop rule [**R-**$\downarrow$]. The following elaborates on how this rule can be utilised to model different types of failure:

- **Message loss** is modelled directly through the reduction rule [**R-**$\downarrow$].
- **Crash-failure** is modelled through repeated applications of [**R-**$\downarrow$] for a particular role. *E.g.*, to model a crash of role p, the reduction step [**R-**$\downarrow$] should be applied to all messages that enter the buffer matching the pattern $(\mathsf{p} \triangleright \_ \triangleleft \_\langle\_\rangle)$ ($\_$ symbolises a "don't care" value).
- **Link-failure** is modelled using a similar method; the difference being that messages between *two* specific recipients are dropped. *E.g.*, modelling a link-failure between roles p and q requires [**R-**$\downarrow$] to be applied to all messages entering the buffer with the patterns $(\mathsf{p} \triangleright \mathsf{q} \triangleleft \_\langle\_\rangle)$ and $(\mathsf{q} \triangleright \mathsf{p} \triangleleft \_\langle\_\rangle)$.
- **Message delay** is modelled by applying rule [**R-**⏲] to a branch whilst a valid message resides in the buffer. *E.g.*:

$$\begin{array}{ll} s[\mathsf{q}]\, \&_{i\in I}\{[\mathsf{p}_i]\,?\,\mathsf{m}_i(x_i).P_i,\ ⏲.\,Q\} \;\mid\; s : (\mathsf{p}_k \triangleright \mathsf{q} \triangleleft \mathsf{m}_k\langle w\rangle) \cdot \sigma & \\ \quad\longrightarrow\;\; Q \;\mid\; s : (\mathsf{p}_k \triangleright \mathsf{q} \triangleleft \mathsf{m}_k\langle w\rangle) \cdot \sigma & \text{for } k \in I. \end{array}$$

- Total **message reordering** is modelled via *buffer congruence rules* (def. 3).
- **Network partitions** can be represented using multiple *link failures.*

The granularity at which we model failures allows for degrees of customisation. *E.g.*, benign fault-tolerant consensus algorithms typically assume the possibility of *all* non-Byzantine faults, therefore all the aforementioned failures are required. Alternatively, an application assumed to run over a trusted TCP network need not worry about single message drops, and hence [**R-**$\downarrow$] should only be applied to model node crash and link failures.

Definition 5 (Well-formedness). *To ensure that communication is possible, we require that a well-formed process has a buffer for each session, i.e.,*

$$P = (\nu s)\, Q \implies Q \equiv (\nu \tilde{s'})\,(Q' \mid s : \sigma)$$

Def. 5 introduces a well-formedness condition to guarantee that a session always guards its buffer, hence ensuring that messages always have a queue to be placed in. From now on, we will only consider well-formed processes.

Before concluding this section, we recall our ping pong running example from the introduction, and present below the processes for roles p, q and r.

Example 2 (Ping Pong: Processes).

$$P_{\mathsf{p}} = s[\mathsf{p}] \oplus [\mathsf{q}]\,!\,\mathsf{ping}\langle\rangle.\, s[\mathsf{p}]\,\&\begin{cases}[\mathsf{q}]\,?\,\mathsf{pong}().P^{ok}_{\mathsf{p}},\\ \text{⏲}.\, s[\mathsf{p}] \oplus [\mathsf{q}]\,!\,\mathsf{ping}\langle\rangle.\, s[\mathsf{p}]\,\&\begin{cases}[\mathsf{q}]\,?\,\mathsf{pong}().P^{ok}_{\mathsf{p}},\\ \text{⏲}.\, s[\mathsf{p}] \oplus [\mathsf{q}]\,!\,\mathsf{ping}\langle\rangle.\, P'_{\mathsf{p}}\end{cases}\end{cases}$$

$$P^{ok}_{\mathsf{p}} = s[\mathsf{p}] \oplus [\mathsf{r}]\,!\,\mathsf{ok}\langle\rangle.\,\mathbf{0} \qquad P'_{\mathsf{p}} = s[\mathsf{p}]\,\&\begin{cases}[\mathsf{q}]\,?\,\mathsf{pong}().P^{ok}_{\mathsf{p}},\\ \text{⏲}.\, s[\mathsf{p}] \oplus [\mathsf{r}]\,!\,\mathsf{ko}\langle\rangle.\,\mathbf{0}\end{cases}$$

$$P_{\mathsf{q}} = s[\mathsf{q}]\,\&\begin{cases}[\mathsf{p}]\,?\,\mathsf{ping}().P^{pong}_{\mathsf{q}},\\ \text{⏲}.\, s[\mathsf{q}]\,\&\begin{cases}[\mathsf{p}]\,?\,\mathsf{ping}().P^{pong}_{\mathsf{q}},\\ \text{⏲}.\, s[\mathsf{q}]\,\&\{[\mathsf{p}]\,?\,\mathsf{ping}().P^{pong}_{\mathsf{q}},\ \text{⏲}.\,\mathbf{0}\}\end{cases}\end{cases}$$

$$P^{pong}_{\mathsf{q}} = s[\mathsf{q}] \oplus [\mathsf{p}]\,!\,\mathsf{pong}\langle\rangle.\,\mathbf{0}$$

$$P_{\mathsf{r}} = s[\mathsf{r}]\,\&\{[\mathsf{p}]\,?\,\mathsf{ok}().\mathbf{0}, [\mathsf{p}]\,?\,\mathsf{ko}().\mathbf{0}\}$$

3 MAGπ: Type System

We introduce the type system for MAGπ, which is a conservative extension of the *generalised* asynchronous MPST theory [33, sec. 7]. *Generalised* MPST stray away from global protocol specifications (global types) and instead operate on user-defined localised specifications of each participating role (local types). The benefits of working with such theory include: *(i)* the ability to capture a larger set of viable protocols compared to traditional syntactic methods (*e.g.* global types) of enforcing consistent communication; *(ii)* the ability to model protocols of different requirements. In particular, instead of syntactically enforcing programmers to write, *e.g.*, deadlock-free code, a generalised theory allows programmers to unrestrictedly design protocols that are checked *a posteriori* against any number of required properties, such as deadlock-freedom, termination *etc.*

Basic and Session Types

$$\mathbb{T} ::= \mathbb{B} \mid \mathbb{S}$$

$$\mathbb{B} ::= \texttt{int} \mid \texttt{bool} \mid \texttt{real} \mid \texttt{unit} \mid \cdots$$

$$\begin{aligned}\mathbb{S} ::=\ & \&_{i\in I}\{\mathsf{p}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i[,\ \circlearrowleft.\,\mathbb{S}']\}\\ \mid\ & \oplus_{i\in I}\{\mathsf{p}_i\,!\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\}\\ \mid\ & \mu\mathrm{t}.\mathbb{S} \mid \mathrm{t} \mid \textbf{end}\end{aligned}$$

Buffer Types

$$\mathbb{M} ::= \mathsf{p}\,!\,\mathsf{m}(\mathbb{T})\cdot\mathbb{M} \mid \epsilon$$

Session-Buffer Types

$$\tau ::= \mathbb{M} \mid \mathbb{S} \mid (\mathbb{M}\,;\mathbb{S})$$

Fig. 3. Basic Types, Session Types, Buffer Types and Session-Buffer Types

$$\frac{}{\mathbb{T}\equiv\mathbb{T}} \qquad \frac{}{\mathbb{M}_1\cdot\mathbb{M}_2\equiv\mathbb{M}_2\cdot\mathbb{M}_1} \qquad \frac{}{\epsilon\cdot\epsilon\equiv\epsilon} \qquad \frac{\mathbb{M}\equiv\mathbb{M}' \qquad \mathbb{S}\equiv\mathbb{S}'}{(\mathbb{M}\,;\mathbb{S})\equiv(\mathbb{M}'\,;\mathbb{S}')}$$

Fig. 4. Type congruence rules

The novelties of our type system include: *(i) undirected branching/selection*; *(ii) timeout branches* (syntax in § 3.1); and *(iii) reliability sets*—sets of roles assumed to not fail, from the perspective of *each* role (§ 3.2). Reliability sets (possibly empty) enforce the use of *timeouts* for all failure-prone communication.

As in [33], our type system does *not* use global types, but solely relies on local types. Consequently, typing contexts must obey a safety property to ensure subject reduction (§ 3.3). Finally, we present the rules for our type system in § 3.4, and discuss its key properties in § 4.

3.1 Types

Our MPST theory is designed for the distributed computing setting. Concretely, our type system (def. 6) is *asynchronous*; it allows *branching* (resp. *selection*) from (resp. to) multiple roles; and supports *timeout* continuation types.

Definition 6 (Typing syntax). *The typing syntax is defined using the grammar in fig. 3. For undirected branching and selection, $I \neq \emptyset$ and role-label tuples $(\mathsf{p}_i, \mathsf{m}_i)$ must be pairwise distinct. Recursion variables cannot be free and must appear guarded under branching/selection types.*

Type $\mathbb{T}$ denotes either a *basic* type $\mathbb{B}$, or a *session type* $\mathbb{S}$, and is used to type variables. Session types describe how a channel should be used: (i) *undirected branching* (external choice) $\&_{i\in I}\{\mathsf{p}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i[,\ \circlearrowleft.\,\mathbb{S}']\}$ denotes *receiving* a message with label m_i and payload of type $\mathbb{T}_i$ from role p_i, then continuing according to $\mathbb{S}_i$. The (optional) timeout continuation type $\mathbb{S}'$ describes the protocol for handling failure on that branch; (ii) *undirected selection* (internal choice) $\oplus_{i\in I}\{\mathsf{p}_i\,!\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\}$ denotes *sending* a message with label m_i and payload $\mathbb{T}_i$ to role p_i, then continuing according to $\mathbb{S}_i$; (iii) type **end** marks a channel as closed, and terminates communication. A session buffer is typed using the *buffer* *type* $\mathbb{M}$. Entries in the buffer must correspond to the type $\mathsf{p}\,!\,\mathsf{m}(\mathbb{T}){\cdot}\mathbb{M}$, denoting a message sent to p with label m and payload of type $\mathbb{T}$. A session with role is typed using *session-buffer* types, combining a session type and a buffer type.

Type *congruence* $\equiv$ is defined in fig. 4. Notably, buffer types can be reordered, and two session-buffer types are congruent if their individual buffer and session types are congruent. Buffer type reordering is necessary to match the *total message reordering* supported by the language (def. 3).

3.2 Reliability

We go on a short detour and talk about reliability. Previous related work [4,1,38] have included the notion of *reliability* into their type systems. Generally, either one specific role, or a pre-defined set of roles, are assumed to be reliable—*i.e.,* no failures occur for communication involving the identified set of roles.

Our definition of reliability (def. 7) is the most general and the first to take into account the viewpoint of each role. We argue that this is necessary in a distributed setting since reliability in networks is dependant on the physical topology of processes. Recalling the ping utility (example 1), we could imagine the processes representing roles p and r reside on the same physical hardware, thus their communication cannot be affected by network faults; and the process for q resides on geographically separated hardware, therefore its communication with both p and r is vulnerable to failure.

Definition 7 (Reliability). *The* ***reliability set*** $\mathfrak{R}$ *for a role* $\mathsf{p} \in \boldsymbol{\rho}$ *is defined as* $\mathfrak{R} \subseteq \boldsymbol{\rho} \setminus \{\mathsf{p}\}$*, capturing the viewpoint of* p*.* ***Reliability*** $\mathcal{R}$ *is defined as a function mapping roles to their reliability set, i.e.,* $\mathcal{R} : \boldsymbol{\rho} \to \mathfrak{R}$*.*

To better model real distributed environments, our definition of reliability allows each role to have its own (possibly empty) reliability set.

Example 3 (Ping Pong: Reliability Sets). W.r.t. example 1, as the three roles have different viewpoints on each other, then the reliability set for each of them is different. In particular, we have $\mathcal{R}(\mathsf{p}) = \{\mathsf{r}\},\ \mathcal{R}(\mathsf{r}) = \{\mathsf{p}\},\ \mathcal{R}(\mathsf{q}) = \emptyset$.

Investigating the extremes, we have: for a set of roles $\boldsymbol{\rho}$, if for all $\mathsf{p} \in \boldsymbol{\rho} \cdot \mathcal{R}(\mathsf{p}) = \emptyset$, then *no* communication is reliable; conversely, if for all $\mathsf{p} \in \boldsymbol{\rho} \cdot \mathcal{R}(\mathsf{p}) = \boldsymbol{\rho} \setminus \{\mathsf{p}\}$, then *all* communication is reliable—referred to as a *reliable network.* This work only considers static configurations for $\mathcal{R}$, thus reliability sets cannot change at runtime. We find that even with this restriction, our definition is the most general compared to related work.

3.3 Contexts

Definition 8 (Type contexts). *Context* Θ *is a partial mapping from process variables to* n*-tuples of types and context* Γ *is a partial mapping from variables to types, and sessions with roles to session-buffer types, both defined below:*

$$\Theta ::= \emptyset \mid \Theta, X : \mathbb{T}_1, \ldots, \mathbb{T}_n \qquad\qquad \Gamma ::= \emptyset \mid \Gamma, x : \mathbb{T} \mid \Gamma, s[\mathsf{p}] : \tau$$

*The **composition** of contexts (Γ_1, Γ_2) is defined iff:*

$$\forall c \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) : \quad \Gamma_i(c) = \mathbb{M} \ \wedge \ \Gamma_j(c) = \mathbb{S}$$

For such c, $(\Gamma_1,\Gamma_2)(\mathrm{c}) = (\mathbb{M}\,;\mathbb{S})$.
*Contexts are **congruent** $\Gamma_1 \equiv \Gamma_2$ iff:*

$$\mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_2) \ \wedge \ \forall c \in \mathsf{dom}(\Gamma_1) : \Gamma_1(c) \equiv \Gamma_2(c)$$

Context Θ is *non-linear* and types *process variables* by tracking the types of their parameters. Context Γ is *linear* and allows *variables* to have *basic* or *session types*, and *sessions with roles* to have *session-buffer types*; as a program progresses, a role may simultaneously have both an active session type and messages queued in the message buffer.

Context *composition* allows two contexts to coexist as long as their common channels map to buffer types in one context, and session types in the other.

Context *congruence* holds if two contexts have the same domain and the types of their channels are congruent. It is key to note that by the definitions of context composition and congruence we have $s[\mathsf{p}] : (\mathbb{M};\, \mathbb{S}) \equiv s[\mathsf{p}] : \mathbb{M}, s[\mathsf{p}] : \mathbb{S}$. Buffer types (resp. session-buffer types) are only used internally by the type system; end-users are not expected to explicitly define these types.

Definition 9 (Context reduction). *An **action** α is given as:*

$$\alpha \ ::= \ s[\mathsf{p}]\,!\,\mathsf{q} : \mathsf{m}(\mathbb{T}) \ \mid \ s[\mathsf{p}][\mathsf{q}] : \mathsf{m} \ \mid \ s[\mathsf{p}]⏲$$

*From left to right, this reads as (i) a **sent message;** (ii) **communication** of a message; and (iii) the **timeout** of a channel. **Context transition** $\xrightarrow{\alpha}_{(\Sigma;\mathcal{R})}$ is defined in fig. 5. We write $\Gamma \xrightarrow{\alpha}_{(\Sigma;\mathcal{R})}$ iff $\exists\, \Gamma' : \Gamma \xrightarrow{\alpha}_{(\Sigma;\mathcal{R})} \Gamma'$. We define two **context reductions** $\rightarrow_{(\Sigma;\mathcal{R})}$ and $\rightarrow_{\Sigma}$ as:*

$$\Gamma \rightarrow_{(\Sigma;\mathcal{R})} \Gamma' \textit{ holds iff } \Gamma \xrightarrow{\alpha}_{(\Sigma;\mathcal{R})} \Gamma'$$

$$\Gamma \rightarrow_{\Sigma} \Gamma' \textit{ holds iff } \Gamma \xrightarrow{\alpha}_{\Sigma} \Gamma' \textit{ for } \alpha \in \{s[\mathsf{p}]\,!\,\mathsf{q} : \mathsf{m}(\mathbb{T}),\ s[\mathsf{p}][\mathsf{q}] : \mathsf{m}\}$$

We write $\rightarrow^{+}_{(\Sigma;\mathcal{R})}$ (resp. $\rightarrow^{+}_{\Sigma}$) and $\rightarrow^{*}_{(\Sigma;\mathcal{R})}$ (resp. $\rightarrow^{*}_{\Sigma}$) for their transitive and reflexive/transitive closures respectively.

A context Γ keeps track of open buffers using a *buffer-tracker* Σ. Whenever a new session is initialised, it is added to Σ, details in § 3.4 item [T-ν]. For now it suffices to know that buffer trackers restrict communication to occur only over restricted sessions, thus by def. 5 (well-formedness), it guarantees that a session buffer exists for all sessions in Σ.

Context reduction (def. 9) models communication at the *type-level.* Context Γ can reduce by sending, communicating, or timing out. By [Γ-⏲], $\Gamma = s[\mathsf{p}] : \&_{i\in I}\{\mathsf{q}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i,\ ⏲.\,\mathbb{S}'\}$ can reduce to a *timeout* branch continuation type $\mathbb{S}'$ if s is in the buffer-tracker (*i.e.,* a buffer exists for session s), and *at least one*

$$[\Gamma\text{-}\circlearrowleft]\ \frac{s \in \Sigma \qquad \exists\, k \in I : \mathsf{q}_k \notin \mathcal{R}(\mathsf{p})}{s[\mathsf{p}] : \&_{i\in I}\{\mathsf{q}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i,\ \circlearrowleft.\mathbb{S}'\} \xrightarrow{s[\mathsf{p}]\circlearrowleft}_{(\Sigma;\mathcal{R})} s[\mathsf{p}] : \mathbb{S}'}$$

$$[\Gamma\text{-Snd}_1]\ \frac{s \in \Sigma \qquad k \in I}{s[\mathsf{p}] : \oplus_{i\in I}\{\mathsf{q}_i\,!\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\} \xrightarrow{s[\mathsf{p}]\,!\,\mathsf{q}_k:\mathsf{m}_k(\mathbb{T}_k)}_{(\Sigma;\mathcal{R})} s[\mathsf{p}] : (\mathsf{q}_k\,!\,\mathsf{m}_k(\mathbb{T}_k)\cdot\epsilon\,;\,\mathbb{S}_k)}$$

$$[\Gamma\text{-Snd}_2]\ \frac{s \in \Sigma \qquad k \in I}{s[\mathsf{p}] : (\mathbb{M}\,;\,\oplus_{i\in I}\{\mathsf{q}_i\,!\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\}) \xrightarrow{s[\mathsf{p}]\,!\,\mathsf{q}_k:\mathsf{m}_k(\mathbb{T}_k)}_{(\Sigma;\mathcal{R})} s[\mathsf{p}] : (\mathbb{M}\cdot\mathsf{q}_k\,!\,\mathsf{m}_k(\mathbb{T}_k)\cdot\epsilon\,;\,\mathbb{S}_k)}$$

$$[\Gamma\text{-Com}]\ \frac{s \in \Sigma \qquad \exists\, k \in I : (\mathsf{p},\mathsf{m},\mathbb{T}) = (\mathsf{p}_k,\mathsf{m}_k,\mathbb{T}_k)}{s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T})\cdot\mathbb{M},\ s[\mathsf{q}] : \&_{i\in I}\{\mathsf{p}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\ [,\circlearrowleft.\mathbb{S}']\} \xrightarrow{s[\mathsf{p}][\mathsf{q}]:\mathsf{m}}_{(\Sigma;\mathcal{R})} s[\mathsf{p}] : \mathbb{M},\ s[\mathsf{q}] : \mathbb{S}_k}$$

$$[\Gamma\text{-}\mu]\ \frac{s[\mathsf{p}] : \mathbb{S}[^{\mu\mathrm{t}.\mathbb{S}}/_{\mathrm{t}}] \xrightarrow{\alpha}_{(\Sigma;\mathcal{R})} \Gamma'}{s[\mathsf{p}] : \mu\mathrm{t}.\mathbb{S} \xrightarrow{\alpha}_{(\Sigma;\mathcal{R})} \Gamma'} \qquad [\Gamma\text{-Cong}]\ \frac{\Gamma_1 \xrightarrow{\alpha}_{(\Sigma;\mathcal{R})} \Gamma_2}{\Gamma,\Gamma_1 \xrightarrow{\alpha}_{(\Sigma;\mathcal{R})} \Gamma,\Gamma_2}$$

Fig. 5. Context reduction rules

of the roles in the branch is *unreliable*. The latter prevents taking a timeout for communication that is sure to be delivered. Reductions [Γ-Snd$_1$] and [Γ-Snd$_2$] simulate sending a message by reducing the selection type $\oplus_{i\in I}\{\mathsf{q}_i\,!\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\}$ to one of its continuations $\mathbb{S}_i$, and by inserting the sent message into the buffer type. The difference is that [Γ-Snd$_1$] creates the buffer type if it was previously not specified, whereas [Γ-Snd$_2$] appends the message to an already existing buffer type. Communication between two roles is simulated through [Γ-Com], where a branch type $s[\mathsf{q}] : \&_{i\in I}\{\mathsf{p}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\ [,\circlearrowleft.\mathbb{S}']\}$ consumes the message from a buffer type $s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T})\cdot\mathbb{M}$, reducing to the continuations $s[\mathsf{p}] : \mathbb{M},\ s[\mathsf{q}] : \mathbb{S}_k$. Lastly, [$\Gamma$-$\mu$] allows reduction through recursion and [Γ-Cong] reduces substructures of compatibly composed contexts.

Definition 10. *Property φ_s is a $(\Sigma;\mathcal{R})$-**safety** property on typing contexts iff:*

[S-$\mathcal{R}_1$] $\varphi_\mathsf{s}(\Gamma,\ s[\mathsf{p}] : \&_{i\in I}\{\mathsf{q}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\}) \implies \forall i \in I : \mathsf{q}_i \in \mathcal{R}(\mathsf{p})$

[S-$\mathcal{R}_2$] $\varphi_\mathsf{s}(\Gamma,\ s[\mathsf{p}] : \&_{i\in I}\{\mathsf{q}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i,\ \circlearrowleft.\mathbb{S}'\}) \implies \exists i \in I : \mathsf{q}_i \notin \mathcal{R}(\mathsf{p})$

[S-Com] $\varphi_\mathsf{s}(\Gamma,\ s[\mathsf{p}] : \&_{i\in I}\{\mathsf{q}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\ [,\ \circlearrowleft.\mathbb{S}']\},\ s[\mathsf{q}]:\mathbb{M})$
and $\mathbb{M} \equiv \mathsf{p}\,!\,\mathsf{m}(\mathbb{T})\cdot\mathbb{M}'$
and $\exists\, k \in I : \mathsf{q}_k = \mathsf{q} \wedge \mathsf{m}_k = \mathsf{m} \implies \mathbb{T}_k = \mathbb{T}$

[S-μ] $\varphi_\mathsf{s}(\Gamma,\ s[\mathsf{p}] : \mu\mathrm{t}.\mathbb{S}) \implies \varphi_\mathsf{s}(\Gamma,\ s[\mathsf{p}] : \mathbb{S}[^{\mu\mathrm{t}.\mathbb{S}}/_{\mathrm{t}}])$

[S-$\rightarrow$] $\varphi_\mathsf{s}(\Gamma)$ and $\Gamma \rightarrow_{(\Sigma;\mathcal{R})} \Gamma' \implies \varphi_\mathsf{s}(\Gamma')$

As previously mentioned, our type system is a *generic* one that does not use syntactic methods of enforcing consistent communication. Therefore, we define a *safety* property in def. 10 on type contexts that is used to guarantee subject reduction and other theorems (presented in § 4).

We say $\varphi_{\mathtt{s}}$ is the *largest* safety property required to guarantee subject reduction. The property can be re-instantiated with more specific conditions (as demonstrated in § 5) as per the requirements of the implementation. Concretely, [S-$\mathcal{R}_1$] and [S-$\mathcal{R}_2$] ensure that timeouts are only not defined if communication is reliable and that timeouts are defined if communication is unreliable respectively. Condition [S-Com] ensures that communicating messages have matching payload types. Lastly, [S-μ] preserves $\varphi_{\mathtt{s}}$ through recursion unfolding and [S-$\rightarrow$] requires safety to hold after context reduction.

3.4 Typing Rules

Our type system is defined by the typing rules in fig. 6. Below we explain them in detail. Typing judgements are of the form: $\Theta \cdot \Gamma \vdash P$ reading "process P is well typed under type contexts Θ and Γ"; and $\Gamma \vdash d : \mathbb{T}$ reading "value (or variable, or channel) d is of type $\mathbb{T}$ under type context Γ".

[T-**0**] The inaction process **0** is typed by a context that is "`end` typed", determined by the predicate $\mathtt{end}(\Gamma)$—defined in fig. 7. The predicate holds: (*i*) if $\Gamma = \emptyset$; (*ii*) if Γ consists of variables, then it holds if all the variables are either of a `basic` type, or can be typed by **end**; and (*iii*) if Γ consists of sessions with roles, then it holds if all the channels can be typed by **end**.

[T-Var] A variable or session with role c has type $\mathbb{T}$ in a context only containing the mapping of c to $\mathbb{T}$.

[T-Val] A value v is typed by a `basic` type $\mathbb{B}$ if v is contained in the set of that `basic` type. *E.g.*, $42 : \mathbb{N}$ is typed under an empty context $\emptyset$ since $42 \in \mathbb{N}$.

[T-X] A process variable X is typed to an n-tuple of types $\mathbb{T}_1, \ldots, \mathbb{T}_n$ under context Θ, if Θ maps the process variable to the same n-tuple of types.

[T-$\oplus$] The *selection* process $c \oplus [\mathbf{q}_k]\,!\,\mathsf{m}_k\langle d\rangle.\,P$ is typed under a context which maps the sending channel c to a selection session type $\oplus_{i\in I}\{\mathbf{q}_i\,!\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\}$, where a selection option matches the send process, *i.e.*, $k \in I$. The context should match the payload d to the type indicated in the selection ($\mathbb{T}_k$), and continuation process P should be typed under the continuation type $\mathbb{S}_k$.

[T-&] The *branching* process $c\,\&_{i\in I}\{[\mathbf{p}_i]\,?\,\mathsf{m}_i(x_i).P_i\}$ is typed under a context which maps the receiving channel c to a branch type $\&_{i\in I}\{\mathbf{p}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\}$, where all roles and message labels of each branch match. Every continuation process P_i must be typed under the continuation type $\mathbb{S}_i$ and payload typed by $\mathbb{T}_i$. If the process is a *timeout* branch $c\,\&_{i\in I}\{[\mathbf{p}_i]\,?\,\mathsf{m}_i(x_i).P_i,\ ⏲.\,Q\}$, then it should be typed by a session type also containing a timeout continuation $\&_{i\in I}\{\mathbf{p}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i,\ ⏲.\,\mathbb{S}'\}$, and the timeout process Q must be typed by $\mathbb{S}'$.

[T-Call] A process *call* $X\langle d_1, \ldots, d_n\rangle$ is correctly typed if Θ types the process variable to a n-tuple of types $\mathbb{T}_1, \ldots, \mathbb{T}_n$ and Γ maps each parameter d_i to the corresponding $\mathbb{T}_i$ (for $i \in 1..n$). Any remaining channels in Γ cannot be open, and hence must be `end` typed.

[T-**0**]
$$\frac{\mathsf{end}(\Gamma)}{\Theta \cdot \Gamma \vdash \mathbf{0}}$$

[T-Var]
$$\frac{}{c : \mathbb{T} \vdash c : \mathbb{T}}$$

[T-Val]
$$\frac{v \in \mathbb{B}}{\emptyset \vdash v : \mathbb{B}}$$

[T-X]
$$\frac{\Theta(X) = \mathbb{T}_1, \ldots, \mathbb{T}_n}{\Theta \vdash X : \mathbb{T}_1, \ldots, \mathbb{T}_n}$$

[T-$\oplus$]
$$\frac{\Gamma_1 \vdash c : \oplus_{i \in I}\{\mathsf{q}_i \,!\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\} \qquad k \in I \qquad \Gamma_2 \vdash d : \mathbb{T}_k \qquad \Theta \cdot \Gamma, c : \mathbb{S}_k \vdash P}{\Theta \cdot \Gamma, \Gamma_1, \Gamma_2 \vdash c \oplus [\mathsf{q}_k] \,!\, \mathsf{m}_k\langle d \rangle .\, P}$$

[T- &]
$$\frac{\begin{array}{c}\Gamma' \vdash c : \&_{i \in I}\{\mathsf{p}_i \,?\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\,[,\ \text{⏲}.\,\mathbb{S}']\} \\ [\Theta \cdot \Gamma, c : \mathbb{S}' \vdash Q] \qquad \forall i \in I \cdot \Theta \cdot \Gamma, x_i : \mathbb{T}_i, c : \mathbb{S}_i \vdash P_i\end{array}}{\Theta \cdot \Gamma, \Gamma' \vdash c \,\&_{i \in I}\{[\mathsf{p}_i] \,?\, \mathsf{m}_i(x_i).P_i\,[,\ \text{⏲}.\,Q]\}}$$

[T-Call]
$$\frac{\Theta \vdash X : \mathbb{T}_1, \ldots, \mathbb{T}_n \qquad \mathsf{end}(\Gamma') \qquad \forall i \in 1..n \cdot \Gamma_i \vdash d_i : \mathbb{T}_i}{\Theta \cdot \Gamma_1, \ldots, \Gamma_n, \Gamma' \vdash X\langle d_1, \ldots, d_n \rangle}$$

[T-Def]
$$\frac{\Theta, X : \mathbb{T}_1, \ldots, \mathbb{T}_n \cdot x_1 : \mathbb{T}_1, \ldots, x_n : \mathbb{T}_n \vdash P \qquad \Theta, X : \mathbb{T}_1, \ldots, \mathbb{T}_n \cdot \Gamma \vdash Q}{\Theta \cdot \Gamma \vdash \mathsf{def}\ X(x_1 : \mathbb{T}_1, \ldots, x_n : \mathbb{T}_n) = P\ \mathsf{in}\ Q}$$

[T-+]
$$\frac{\Theta \cdot \Gamma \vdash P_1 \qquad \Theta \cdot \Gamma \vdash P_2}{\Theta \cdot \Gamma \vdash P_1 + P_2}$$

[T-Lift]
$$\frac{\Theta \cdot \Gamma \vdash P}{\Theta \cdot \Gamma \vdash_{\emptyset} P}$$

[T-ϵ]
$$\frac{\mathsf{gc}(\Gamma)}{\Theta \cdot \Gamma \vdash_{\{s\}} s : \epsilon}$$

[T-σ_1]
$$\frac{\Theta \cdot \Gamma' \vdash_{\{s\}} s : \sigma \qquad \Gamma \vdash w : \mathbb{T}}{\Theta \cdot \Gamma, \Gamma', s[\mathsf{p}] : \mathsf{q} \,!\, \mathsf{m}(\mathbb{T}) \cdot \epsilon \vdash_{\{s\}} s : (\mathsf{p} \triangleright \mathsf{q} \triangleleft \mathsf{m}\langle w \rangle) \cdot \sigma}$$

[T-σ_2]
$$\frac{\Theta \cdot \Gamma', s[\mathsf{p}] : \mathbb{M} \vdash_{\{s\}} s : \sigma \qquad \Gamma \vdash w : \mathbb{T}}{\Theta \cdot \Gamma, \Gamma', s[\mathsf{p}] : \mathsf{q} \,!\, \mathsf{m}(\mathbb{T}) \cdot \mathbb{M} \vdash_{\{s\}} s : (\mathsf{p} \triangleright \mathsf{q} \triangleleft \mathsf{m}\langle w \rangle) \cdot \sigma}$$

[T-σ_w]
$$\frac{\Gamma = (\Gamma_0 \rightsquigarrow \Gamma_1), \Gamma_2 \qquad \Theta \cdot \Gamma_1 \vdash_{\Sigma} s : \sigma \qquad \mathsf{gc}(\Gamma_0, \Gamma_2)}{\Theta \cdot \Gamma \vdash_{\Sigma} s : \sigma}$$

[T-|]
$$\frac{\Theta \cdot \Gamma_1 \vdash_{\Sigma_1} P_1 \qquad \Theta \cdot \Gamma_2 \vdash_{\Sigma_2} P_2 \qquad \Sigma_1 \cap \Sigma_2 = \emptyset}{\Theta \cdot \Gamma_1, \Gamma_2 \vdash_{\Sigma_1 \cup \Sigma_2} P_1 \mid P_2}$$

[T-ν]
$$\frac{\Gamma' = \{s[\mathsf{p}] : \tau_{\mathsf{p}}\}_{\mathsf{p} \in \boldsymbol{\rho}} \qquad s \notin \Gamma \qquad (\{s\}\,;\mathcal{R})\text{-}\varphi_{\mathsf{s}}(\Gamma') \qquad \Theta \cdot \Gamma, \Gamma' \vdash_{\Sigma} P}{\Theta \cdot \Gamma \vdash_{\Sigma \setminus \{s\}} (\nu s {:} \Gamma')\, P}$$

Fig. 6. Typing rules

$$\frac{}{\texttt{end}(\emptyset)} \qquad \frac{\forall i \in 1..n \;\cdot\; \texttt{basic}(\mathbb{T}_i) \;\vee\; x_i : \mathbb{T}_i \;\vdash\; x_i : \mathbf{end}}{\texttt{end}(x : \mathbb{T}_1, \ldots, x_n : \mathbb{T}_n)}$$

$$\frac{\texttt{end}(\Gamma_1) \qquad \texttt{end}(\Gamma_2)}{\texttt{end}(\Gamma_1, \Gamma_2)} \qquad \frac{\forall i \in 1..n, \mathsf{p} \in \boldsymbol{\rho} \;\cdot\; s_i[\mathsf{p}] : \tau_i \;\vdash\; s_i[\mathsf{p}] : \mathbf{end}}{\texttt{end}(s_i[\mathsf{p}] : \tau_1, \ldots, s_i[\mathsf{p}] : \tau_n)}$$

Fig. 7. Predicate $\texttt{end}(\Gamma)$

$$\frac{}{\texttt{gc}(\emptyset)} \qquad \frac{\texttt{gc}(\Gamma)}{\texttt{gc}(\Gamma, s[\mathsf{p}] : \epsilon)} \qquad \frac{\texttt{basic}(\mathbb{T}) \qquad \texttt{gc}(\Gamma, s[\mathsf{p}] : \mathbb{M})}{\texttt{gc}(\Gamma, s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T}) \cdot \mathbb{M})}$$

$$\frac{\Gamma = \Gamma', s'[\mathsf{p}'] : \mathbb{T} \qquad \texttt{gc}(\Gamma', s[\mathsf{p}] : \mathbb{M})}{\texttt{gc}(\Gamma, s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T}) \cdot \mathbb{M})}$$

Fig. 8. The garbage collector predicate $\texttt{gc}(\Gamma)$

$$s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T}) \cdot \epsilon \rightsquigarrow \Gamma, s[\mathsf{p}] : \mathbb{M} = \Gamma, s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T}) \cdot \mathbb{M}$$
$$s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T}) \cdot \epsilon \rightsquigarrow \Gamma \;\textit{ when }\; s[\mathsf{p}] : \mathbb{M} \notin \Gamma = \Gamma, s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T}) \cdot \epsilon$$

Fig. 9. Message insertion function $\Gamma' \rightsquigarrow \Gamma$

[T-Def] Process *declaration* $X(x_1 : \mathbb{T}_1, \ldots, x_n : \mathbb{T}_n) = P$ is well typed if P is self-contained, *i.e.,* contexts containing the types of the declaration parameters (along with any previous Θ) should type P. Process *definition* $\mathsf{def}\ X(x_1 : \mathbb{T}_1, \ldots, x_n : \mathbb{T}_n) = P\ \mathsf{in}\ Q$ is typed under $\Theta \cdot \Gamma$ if its declaration is well typed and Q is typed under Γ and Θ composed with the new process variable.

[T-+] Non-deterministic choice is well typed if processes are typed by $\Theta \cdot \Gamma$ in isolation. This is in line with how *case* or *if-then-else* processes are typed.

[T-Lift] We annotate the typing judgement $\Theta \cdot \Gamma \vdash P$ with the buffer-tracker to obtain $\Theta \cdot \Gamma \vdash_{\Sigma} P$, denoting that the sessions in Σ occur in P. The lifting rule annotates the typing judgement with an empty buffer-tracker if the buffer-less judgement ($\vdash$) types P (using the rules mentioned thus far).

[T-ϵ] In standard asynchronous MPST theory, the empty buffer $s : \epsilon$ is typed under the empty context $\emptyset$, ensuring a one-to-one correlation between buffer types in the context and messages in a session buffer. However, since our calculus *models message loss*, it is possible that a context contains buffer types for messages that have been dropped from the process buffer. Thus, our theory types $s : \epsilon$ under a *garbage collected* Γ. The predicate $\texttt{gc}$ is defined in fig. 8, and states that valid leftover types $\texttt{gc}(\Gamma)$ are: (i) empty; (ii) empty buffer types; (iii) message buffer types with basic-type payloads; or (iv) message buffer types with channel payloads that are typed under Γ.

[T-σ_1] [T-σ_2] An entry in a session buffer $s : (\mathbf{p} \triangleright \mathbf{q} \triangleleft \mathsf{m}\langle w \rangle) \cdot \sigma$ is typed under a context containing a mapping from $s[\mathbf{p}]$ to a buffer type $\mathbf{q}\,!\,\mathsf{m}(\mathbb{T}){\cdot}\mathbb{M}$, matching the recipient and message label. The message payload w must be of type $\mathbb{T}$, indicated by the buffer type, and buffer continuation $s : \sigma$ should be typed under the buffer continuation type $\mathbb{M}$ in the case that it is not empty ([T-σ_2]).

[T-σ_w] Weakening allows a session buffer to be typed under a larger context if the addition can be garbage collected and inserted into the original context using the message insertion function (fig. 9). This is partial function that either appends a message to an existing buffer type, or inserts it as the head of a new buffer type. Put differently, weakening allows a buffer to be typed under a larger context containing irrelevant types that can be garbage collected.

[T-|] If a process P_1 is typed by Γ_1, and process P_2 is typed by Γ_2, then the composition Γ_1,Γ_2 types the *parallel* composition $P_1 \mid P_2$. It is also required that parallel processes *cannot* each contain a buffer for the same session s. This guarantees the uniqueness of one session buffer per restricted session.

[T-ν] Session restriction $(\nu s : \Gamma')\, P$ requires session s to be instantiated with a Γ' mapping each session with role to its session-buffer type. $\varphi_{\mathsf{s}}(\Gamma')$ must hold to ensure subject reduction, as discussed in § 3.3. Session s should not be present in a previous context Γ, and process P should be typed under the composition of the previous and newly instantiated context with the updated buffer-tracker $\Theta \cdot \Gamma, \Gamma' \ \vdash_{\Sigma} P$ (since the buffer for s is contained in P).

Example 4 (Ping Pong: Type Context). Recalling the ping pong example, the whole system can then be described by a parallel composition of the three processes representing each role $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$ together with an empty buffer, which is closed under a type context Γ with the following typing assumptions.

$$\Gamma = \{s[\mathbf{p}] : \mathbb{S}_{\mathbf{p}},\ s[\mathbf{q}] : \mathbb{S}_{\mathbf{q}}, s[\mathbf{r}] : \mathbb{S}_{\mathbf{r}}\}$$

$$P_{ping} = (\nu s : \Gamma)\ P_{\mathbf{p}} \mid P_{\mathbf{q}} \mid P_{\mathbf{r}} \mid s : \epsilon$$

4 Type Properties

The main results of our MPST system for MAGπ processes are *subject reduction* (theorem 1) and *session fidelity* (theorem 2). It is key to note that our results are parametric on the reliability function $\mathcal{R}$. Thus, the theorems we present hold for *any* configuration of reliability, *i.e.,* from *no* reliable communication all the way to *completely* reliable networks.

In order to synchronise reliability assumptions between types and processes, we define the *reliable process reduction* $\longrightarrow_{\mathcal{R}}$, such that $\longrightarrow_{\mathcal{R}} \subseteq \longrightarrow$.

Definition 11 (Reliable process reduction). *The reliable process reduction $\longrightarrow_{\mathcal{R}}$ is inductively defined by the same reduction rules for $\longrightarrow$ (in fig. 2), with the following changes* [3]*:*

$$[\mathbf{R}\text{-}⏲] \quad s[\mathbf{q}]\,\&_{i\in I}\{[\mathbf{p}_i]\,?\,\mathsf{m}_i(x_i).P_i, ⏲.\,Q\} \mid s:\sigma \longrightarrow_{\mathcal{R}} Q \mid s:\sigma \qquad \text{if } \exists k \in I : \mathbf{p}_k \notin \mathcal{R}(\mathbf{q})$$

$$[\mathbf{R}\text{-}\lightning] \quad s : (\mathbf{p} \triangleright \mathbf{q} \triangleleft \mathsf{m}\langle w \rangle) \cdot \sigma \longrightarrow_{\mathcal{R}} s : \sigma \qquad \text{for } \mathbf{q} \notin \mathcal{R}(\mathbf{p})$$

[3] For a fully unreliable network, *i.e.,* $\forall \mathbf{p} \in \boldsymbol{\rho} \cdot \mathcal{R}(\mathbf{p}) = \emptyset$, $\longrightarrow_{\mathcal{R}}$ is equivalent to $\longrightarrow$.

Intuitively, the reliable process reduction disregards network faults for reliable communication. Concretely, a timeout reduction [**R-**$\circlearrowright$] is only possible if *at least one role* in the branch is unreliable; and message loss [**R-**$\lightning$] can only occur for messages that are *not* reliable from the viewpoint of the sender. This ensures that no messages are ignored or lost for reliable communication. Proofs of our theorems, along with any auxiliary results, are given in the technical report [23].

4.1 Subject Reduction

Using $\longrightarrow_{\mathcal{R}}$, we now present our result of *subject reduction*. Intuitively, subject reduction states that, if a process P is typed under a safe context, and P reliably reduces to some process P', then the context also reduces (in 0 or 1 steps) to a safe context, which types the new process P'.

Theorem 1 (Subject Reduction).

$$\Theta \cdot \Gamma \vdash_{\Sigma} P \;\; and \;\; (\Sigma;\mathcal{R})\text{-}\varphi_{\mathsf{s}}(\Gamma) \;\; and \;\; P \rightarrow_{\mathcal{R}} P' \implies$$
$$\exists \Gamma' : \Gamma \rightarrow^{\{0,1\}}_{(\Sigma;\mathcal{R})} \Gamma' \;\; and \;\; (\Sigma;\mathcal{R})\text{-}\varphi_{\mathsf{s}}(\Gamma') \;\; and \;\; \Theta \cdot \Gamma' \vdash_{\Sigma} P'$$

A key novel result of our type system is that no unexpected network failures can occur at runtime, *i.e.*, a process always has a failure-handling subprotocol defined for unreliable communication. This follows from the definition of our safety property φ_{s} (def. 10) and holds through subject reduction. We state the result in cor. 1. More precisely, this corollary states that timeout branches are guaranteed to be defined for unreliable communication. The inverse is stated in cor. 2, *i.e.*, timeouts are not defined for branches containing *only* reliable sources.

Corollary 1 (Failure handling safety). *Given a reliability function* $\mathcal{R}$: $\mathbf{p} \notin \mathcal{R}(\mathbf{q})$ *and* $\Theta \cdot \Gamma \vdash_{\Sigma} P$ *with* $(\Sigma;\mathcal{R})\text{-}\varphi_{\mathsf{s}}(\Gamma)$ *and* $P \longrightarrow^{*}_{\mathcal{R}} P' \equiv \mathbb{C}[Q]$ *implies* $Q \neq s[\mathbf{q}] \&_{i\in I}\{\ldots, [\mathbf{p}]\,?\,\mathsf{m}(x).Q'\}$. *I.e.,* Q *cannot be a branch at* $\mathbf{q}$ *receiving from* $\mathbf{p}$ *and not define a timeout.*

Corollary 2 (Reliability adherence). *Given a reliability function* $\mathcal{R}$: $\mathcal{R}(\mathbf{q}) = \mathfrak{R}_{\mathbf{q}}$ *and* $\Theta \cdot \Gamma \vdash_{\Sigma} P$ *with* $(\Sigma;\mathcal{R})\text{-}\varphi_{\mathsf{s}}(\Gamma)$ *and* $P \longrightarrow^{*}_{\mathcal{R}} P' \equiv \mathbb{C}[Q]$ *implies* $Q \neq s[\mathbf{q}] \&_{i\in I}\{[\mathbf{p}_i]\,?\,\mathsf{m}_i(x_i).Q_i,\ \circlearrowright.\,Q'\}$ st: $\forall i \in I : \mathbf{p}_i \in \mathfrak{R}_{\mathbf{q}}$. *I.e.,* Q *cannot be a branch at* $\mathbf{q}$ *only receiving from reliable roles* $\mathbf{p}_i$ *and define a timeout.*

4.2 Session Fidelity

Session fidelity states the opposite implication of subject reduction, *i.e.*, if Γ types a process P, and Γ can reduce, then P can match at least one of the context reductions.

Consequently, relevant properties of process P can be deduced from the behaviour of its type context Γ (as we will see in theorem 3). However, as shown by Scalas and Yoshida [33, sec. 5.2], the result does *not* hold for all well-typed processes. Concretely, session fidelity is violated by: (i) processes that recurse infinitely without being productive (*e.g.* $\mathsf{def}\ X(x) = X\langle x\rangle\ \mathsf{in}\ X\langle s[\mathsf{p}]\rangle$); and (ii) processes that deadlock by interleaving communication across multiparty sessions. Hence, we assume the necessary conditions on processes to restrict the aforementioned violations, by adapting [33, def. 5.3].

Definition 12 (Conditions for session fidelity). *Assuming* $\emptyset \cdot \Gamma \vdash_{\{s\}} P$. *We say that* P:

1. ***has guarded definitions*** *iff each process definition in* P *of the form*

$$\mathsf{def}\ X(x_1 : \mathbb{T}, \ldots, x_n : \mathbb{T}) = Q\ \mathsf{in}\ P'$$

$\forall j \in 1..n$: *if* $\mathbb{T}_j$ *is a session type, then a process call* $Y\langle \ldots, x_j, \ldots\rangle$ *can only occur in* Q *as a subterm of*

$$x_j \&_{i\in I}\{[\mathsf{p}_i]\,?\,\mathsf{m}_i(y_i).P_i[,\ \circlearrowleft.\,P_t]\}\quad \text{or}\quad x_j \oplus [\mathsf{p}]\,!\,\mathsf{m}\langle y\rangle.\,P'',$$

i.e., after x_j *is used for input or output.*

2. ***only plays role*** p ***in*** s***, by*** Γ *iff: **(i)*** P *has guarded definitions (from 1); **(ii)*** $\mathtt{fv}(P) = \emptyset$*; **(iii)*** $\Gamma = \Gamma_0, s[\mathsf{p}] : \tau$ *with* $\tau \neq \mathbf{end}$ *and* $\mathtt{end}(\Gamma_0)$*; and **(iv)** for all* $(\nu s' : \Gamma')\, P'$ *subterm of* P, $\mathtt{end}(\Gamma')$.

We say "P *only plays role* p *in* s*" iff* $\exists \Gamma : \emptyset \cdot \Gamma \vdash_{\{s\}} P$ *and condition 2 holds.*

Def. 12 formalises guarded recursion in condition 1, and the notion of only playing a single role for a given session in condition 2. Together, these conditions ensure that session fidelity, stated in theorem 2, holds for all well-typed processes.

Theorem 2 (Session Fidelity). *Assuming* $\emptyset \cdot \Gamma \vdash_\Sigma P$ *with* $(\Sigma;\mathcal{R})$-$\varphi_{\mathsf{s}}(\Gamma)$, $P \equiv (\Pi_{\mathsf{p}\in I}\, P_{\mathsf{p}}) \,|\, s : \sigma$ *and* $\Gamma = \bigcup_{\mathsf{p}\in I} \Gamma_{\mathsf{p}}$*, and for each* P_{p}*: (i)* $\emptyset \cdot \Gamma_{\mathsf{p}} \vdash_\Sigma P_{\mathsf{p}}$*, and (ii)* P_{p} *being* $\mathbf{0}$ *(up-to-*$\equiv$*)* or *only plays role* p *in* s*, by* Γ_{p}*. Then,*

$\Gamma \longrightarrow_{(\Sigma;\mathcal{R})}$ *implies* $\exists \Gamma', P'$*: (i)* $\Gamma \longrightarrow_{(\Sigma;\mathcal{R})} \Gamma'$*, (ii)* $P \longrightarrow^{+}_{\mathcal{R}} P'$*, (iii)* $\emptyset \cdot \Gamma' \vdash_\Sigma P'$ *with* $(\Sigma;\mathcal{R})$-$\varphi_{\mathsf{s}}(\Gamma')$*, (iv)* $P' = (\Pi_{\mathsf{p}\in I}\, P'_{\mathsf{p}}) \,|\, s : \sigma'$ *and* $\Gamma' = \bigcup_{\mathsf{p}\in I} \Gamma'_{\mathsf{p}}$*, and (v) for each* P'_{p}*:* $\emptyset \cdot \Gamma'_{\mathsf{p}} \vdash_\Sigma P'_{\mathsf{p}}$*, and* P'_{p} *is* $\mathbf{0}$ *(up-to-*$\equiv$*)* or *only plays role* p *in* s*, by* Γ'_{p}*.*

4.3 Process Properties

Our result of session fidelity (§ 4.2) allows us to infer runtime properties about programs in MAGπ from their types. We proceed by defining desirable runtime properties on processes (def. 13); expressing the equivalence of these properties at type-level (def. 14); and presenting our result of *process properties verification* (theorem 3), linking process properties to their type-level equivalences.

From def. 13 below, a process is: **(i)** $\mathcal{R}_F$*-communication-safe* (new w.r.t.[33]) if it reaches the end of communication over reliable reductions and has *no leftover messages* in the buffer; **(ii)** *deadlock-free* if it either reduces or it is inaction; **(iii)** *terminating* if it is deadlock free and can reach inaction in a finite number of steps; **(iv)** *never-terminating* if it can always infinitely reduce; and **(v)** *live* if, for every reliable branch it can reduce to, it can eventually reduce to some branch continuation. We need not consider branches with timeouts since these are trivially live, given that a process can always reduce over the timeout.

Definition 13 (Process properties, adapted from [33]**).** *For some reliability function $\mathcal{R}$, and full reliability function $\mathcal{R}_F$, a process P is said to be:*

(i) ***$\mathcal{R}_F$-communication-safe*** *iff*

$$P \longrightarrow^*_{\mathcal{R}_F} P' \not\longrightarrow_{\mathcal{R}_F} \;\; \textit{and} \;\; P' = \mathbb{C}[s:\sigma] \;\; \textit{implies} \;\; \sigma = \epsilon;$$

(ii) ***deadlock-free*** *iff* $P \longrightarrow^*_{\mathcal{R}} P' \not\longrightarrow_{\mathcal{R}}$ *implies* $P' \equiv \mathbf{0}$*;*

(iii) ***terminating*** *iff it is deadlock free, and*

$$\exists\, i \textit{ finite} \text{ st: } \forall n \geq i : P = P_0 \longrightarrow_{\mathcal{R}} P_1 \longrightarrow_{\mathcal{R}} \cdots \longrightarrow_{\mathcal{R}} P_n \;\; \textit{implies} \;\; P_n \not\longrightarrow_{\mathcal{R}} ;$$

(iv) ***never-terminating*** *iff* $P \longrightarrow^*_{\mathcal{R}} P'$ *implies* $P' \longrightarrow_{\mathcal{R}}$*;*

(v) ***live*** *iff* $P \longrightarrow^*_{\mathcal{R}} P' \equiv \mathbb{C}[Q]$ *implies*

$$\textit{if} \;\; Q = c\,\&_{i\in I}\{[\mathbf{q}_i]\,?\,\mathsf{m}_i(x_i).Q'_i\}, \;\; \textit{then} \quad \exists\,\mathbb{C}', k \in I, w : P' \longrightarrow^*_{\mathcal{R}} \mathbb{C}'[\,Q'_k[^{w}/_{x_k}]\,].$$

Note that, differently from other works [4,33], our definition of liveness only speaks about receiving processes, and not sending. Typically, liveness also requires that a sent message—in the case of MAGπ, any message in a session buffer—is always eventually consumed. However, because of the failures that our calculus models, it is possible that a process is live and still have unconsumed messages in the buffer (*e.g.*, as a result of timing out due to a message delay). Additionally, for a $\mathcal{R}_F$*-communication-safe* process it follows that all sent messages are consumed in the reliable case. Hence, the traditional definition of liveness still holds for reliable network configurations, and our new definition provides the largest guarantees possible given the failure assumptions.

We now present the type-level equivalences of the above process properties. For liveness, we generalise to the largest liveness property, as done with safety in def. 10, allowing users to define more fine-grained notions of liveness, if required.

From def. 14 below, a type context is: **(i)** $\mathcal{R}_F$*-communication-safe* if it has no populated buffer types when it can no longer reliably reduce; **(ii)** *deadlock-free* if the reason why it can no longer reduce is because it is `end` typed (and possibly, as a result of network failures, has some leftover types that can be garbage collected); **(iii)** *terminating* if it is deadlock free and can reach the end of the protocol in a finite number of steps; **(iv)** *never-terminating* if it can always infinitely reduce; and **(v)** *live* if, for every reliable branch it can reduce to, there is a series of steps that can reduce to a continuation of that branch.

Definition 14 (Type context properties). *For some reliability function $\mathcal{R}$, a full reliability function $\mathcal{R}_F$, and a set of sessions Σ, we say context Γ is:*

(i) $(\Sigma;\mathcal{R}_F)$***-communication-safe*** *iff*

$$\Gamma \longrightarrow^*_{(\Sigma;\mathcal{R}_F)} \Gamma' \not\longrightarrow_{(\Sigma;\mathcal{R}_F)} \text{ and } s[\mathsf{p}] : \mathbb{M} \in \Gamma' \text{ implies } \mathbb{M} = \epsilon;$$

(ii) $(\Sigma;\mathcal{R})$***-deadlock-free*** *iff*

$$\Gamma \longrightarrow^*_{(\Sigma;\mathcal{R})} \Gamma' \not\longrightarrow_{(\Sigma;\mathcal{R})} \quad \text{implies} \quad \Gamma' = \Gamma'_0, \Gamma'' \text{ st: } \mathtt{end}(\Gamma'_0) \text{ and } \mathtt{gc}(\Gamma'');$$

(iii) $(\Sigma;\mathcal{R})$***-terminating*** *iff it is* $(\Sigma;\mathcal{R})$*-deadlock-free, and* $\exists\, i$ *finite* st:

$$\forall n \geq i : \Gamma = \Gamma_0 \longrightarrow_{(\Sigma;\mathcal{R})} \Gamma_1 \longrightarrow_{(\Sigma;\mathcal{R})} \cdots \longrightarrow_{(\Sigma;\mathcal{R})} \Gamma_n \text{ implies } \Gamma_n \not\longrightarrow_{(\Sigma;\mathcal{R})} ;$$

(iv) $(\Sigma;\mathcal{R})$***-never-terminating*** *iff* $\Gamma \longrightarrow^*_{(\Sigma;\mathcal{R})} \Gamma'$ *implies* $\Gamma' \longrightarrow_{(\Sigma;\mathcal{R})}$*;*

(v) $(\Sigma;\mathcal{R})$***-live*** *iff it obeys some liveness property* $(\Sigma;\mathcal{R})\text{-}\varphi_{\mathtt{L}}$ st:

$$\begin{aligned}
&(\Sigma;\mathcal{R})\text{-}\varphi_{\mathtt{L}}(\Gamma,\ s[\mathsf{p}] : \mathbb{S}) \text{ and } \mathbb{S} = \&_{i\in I}\{\mathsf{q}_i\, ?\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\} \\
&\qquad\qquad \implies \exists \Gamma', k \in I : \Gamma, s[\mathsf{p}] : \mathbb{S} \longrightarrow^*_{(\Sigma;\mathcal{R})} \Gamma', s[\mathsf{p}] : \mathbb{S}_k \\
&(\Sigma;\mathcal{R})\text{-}\varphi_{\mathtt{L}}(\Gamma,\ s[\mathsf{p}] : \mu \mathrm{t}.\mathbb{S}) \implies (\Sigma;\mathcal{R})\text{-}\varphi_{\mathtt{L}}(\Gamma,\ s[\mathsf{p}] : \mathbb{S}[^{\mu \mathrm{t}.\mathbb{S}}/_{\mathrm{t}}]) \\
&(\Sigma;\mathcal{R})\text{-}\varphi_{\mathtt{L}}(\Gamma) \text{ and } \Gamma \rightarrow_{(\Sigma;\mathcal{R})} \Gamma' \implies (\Sigma;\mathcal{R})\text{-}\varphi_{\mathtt{L}}(\Gamma)
\end{aligned}$$

We are now ready to use these type-level equivalent properties to infer behaviours of the processes they type. We present our result in theorem 3 which formally states that, under the same assumptions given in session fidelity (theorem 2), if a process is typed under some type context, and a property holds on that context, then the same property holds for the process itself.

Theorem 3 (Process properties verification). *Assuming:* $\emptyset \cdot \Gamma \vdash_\Sigma P$ *with* $(\Sigma;\mathcal{R})\text{-}\varphi_{\mathtt{s}}(\Gamma)$*,* $P \equiv (\Pi_{\mathsf{p}\in I}\, P_{\mathsf{p}}) \,|\, s : \sigma$ *and* $\Gamma = \bigcup_{\mathsf{p}\in I} \Gamma_{\mathsf{p}}$*. Further, for each* P_{p}*: (i)* $\emptyset \cdot \Gamma_{\mathsf{p}} \vdash_\Sigma P_{\mathsf{p}}$*, and (ii)* $P_{\mathsf{p}} \equiv \mathbf{0}$ or P_{p} *only plays role* p *in* s*, by* Γ_{p}*. Then,* $\forall \phi \in \{\mathcal{R}_F$*-communication-safe, deadlock-free, terminating, never-terminating, live*$\}$*, if* $(\Sigma;\mathcal{R})\text{-}\phi(\Gamma)$*, then* P *is* ϕ*.*

4.4 Decidability

Since MAGπ is Turing-complete, determining the properties listed in def. 13 from *processes* is *undecidable* [5]. A benefit of our generalised theory is that undecidable process properties can be inferred from *decidable* type-level properties.

Theorem 4 (Decidability). *If* $(\Sigma;\mathcal{R})\text{-}\phi(\Gamma)$ *is decidable, then "*$\Theta \cdot \Gamma \vdash_\Sigma P$ *with* $(\Sigma;\mathcal{R})\text{-}\phi(\Gamma)$*" is decidable.*

Our decidability result (theorem 4) states that for any decidable type-level property, type-checking with that property is decidable. However, since MAGπ is *asynchronous*, we have *no* results on decidability of ϕ. On the contrary, as discussed in [33, sec. 7], type-level properties for *asynchronous* type theories are, in some cases, *undecidable*. This is a result of pairing buffer types with session types—which makes the type system Turing-powerful [3, thm. 2.5]. Scalas and Yoshida [33] address this issue through two methods: (i) standard global types produce type contexts that can be captured through a *decidable* ***consistency*** property; and (*ii*) restricting the size of the message buffer to make properties decidable. The former ensures decidability by restricting communication to match the expressivity of global types. For the latter, they show that any type context that remains bound within a finite-sized buffer is decidable (since the type has a finite state transition system representation). In line with their results, we lift their definition of *boundedness*, *i.e.,* a restriction on the size of a buffer, to MAGπ's type system.

Definition 15 (Boundedness, from [33]**).** *We say Γ is $(\Sigma;\mathcal{R})$-**bound**$_k$ iff* $\exists k \in \mathbb{N} : \Gamma \longrightarrow^*_{(\Sigma;\mathcal{R})} \Gamma', s[\mathsf{p}] : \mathbb{M}$ *implies* $|\mathbb{M}| < k$.
*We say Γ is $(\Sigma;\mathcal{R})$-**bounded** iff $\exists k$ finite* : $(\Sigma;\mathcal{R})$-***bound***$_k(\Gamma)$.

Using def. 15, we present our result of decidable bounded properties in theorem 5.

Theorem 5 (Decidable bounded properties). *$(\Sigma;\mathcal{R})$-bound$_k(\Gamma)$ is decidable for all $\Sigma, \mathcal{R}$, and k. Furthermore, if $(\Sigma;\mathcal{R})$-bounded(Γ), then $\forall \phi \in$ {$\mathcal{R}_F$-communication-safe, deadlock-free, terminating, never-terminating, live}, it holds that $(\Sigma;\mathcal{R})$-$\phi(\Gamma)$ is decidable.*

Thus, decidability is guaranteed for all protocols expressible through standard *asynchronous* global type theory, and all protocols that use finite message buffers—now with the benefit of reasoning about and handling network errors!

Example 5 (Ping Pong: Properties). Inspecting the types in example 1 and example 4, we can conclude that $\Gamma = \{s[\mathsf{p}] : \mathbb{S}_\mathsf{p}, s[\mathsf{q}] : \mathbb{S}_\mathsf{q}, s[\mathsf{r}] : \mathbb{S}_\mathsf{r}\}$ is bound$_4$. By theorem 5, Γ is decidable to check for type-level properties. On doing so, we determine that Γ is: (*i*) *safe*, it satisfies the safety property (def. 10) required for subject reduction; (*ii*) $\mathcal{R}_F$*-communication-safe*, since if we only consider reliable reductions, no buffer types remain populated; (*iii*) *terminating*, since we can count the number of steps taken to reach the end of the protocol; and (*iv*) *live*, as reliable communication $\mathbb{S}_\mathsf{r}$ always reduces—*i.e.,* a result is always obtained.

5 Generalising Network Assumptions

The work presented thus far covers worst-case network assumptions for communication. As beneficial as this may be for low-level networks programming, and for complex distributed applications with minimal assumptions (*e.g.* consensus protocols), not all applications are built on these pessimistic conditions. *E.g.* many distributed applications operate over the Transmission Control Protocol (TCP), and thus assume that if consecutive messages are received from the same source, then they are guaranteed to arrive in the order in which they were sent.

We now showcase the few changes to MAGπ required to alter its network assumptions. It is key to note that these changes produce a *subset* of MAGπ, thus all relevant properties continue to be valid for its TCP-compliant version.

5.1 From Total to Partial Reordering

In a *reliable* network configuration designed to run over TCP, message reordering for communication between *two parties* is guaranteed to *not occur*. Therefore, we can adjust the message reordering of MAGπ to model this environment, and strengthen our safety property $\varphi_{\mathtt{s}}$ to *TCP-safe* communication. MAGπ models message reordering through buffer congruence rules. Therefore, strengthening congruence suffices to restrict communication to the TCP-safe assumptions.

Definition 16 (TCP process-congruence). *The process congruence for the TCP-compliant subset of MAGπ, $\equiv_{\mathit{TCP}}$, is inductively defined using the same rules defining $\equiv$ (in def. 3), but with the following change:*

$$s:\sigma_1 \cdot h_1 \cdot h_2 \cdot \sigma_2 \;\equiv\; s:\sigma_1 \cdot h_2 \cdot h_1 \cdot \sigma_2$$

replaced by

$$\frac{\mathsf{p}_1 \neq \mathsf{p}_2 \;\textit{or}\; \mathsf{q}_1 \neq \mathsf{q}_2}{\begin{array}{c} s:\sigma_1 \cdot (\mathsf{p}_1 \triangleright \mathsf{q}_1 \triangleleft \mathsf{m}_1\langle w_1\rangle) \cdot (\mathsf{p}_2 \triangleright \mathsf{q}_2 \triangleleft \mathsf{m}_2\langle w_2\rangle) \cdot \sigma_2 \\ \equiv_{\mathtt{TCP}} \;\; s:\sigma_1 \cdot (\mathsf{p}_2 \triangleright \mathsf{q}_2 \triangleleft \mathsf{m}_2\langle w_2\rangle) \cdot (\mathsf{p}_1 \triangleright \mathsf{q}_1 \triangleleft \mathsf{m}_1\langle w_1\rangle) \cdot \sigma_2 \end{array}}$$

To obtain the TCP-compliant subset of MAGπ, we assume reductions over fully reliable networks and adopt TCP process congruence from def. 16, which no longer allows reordering of messages for each role couple. We now reflect this definition of TCP congruence at the type-level in def. 17, and use this to define a TCP-safety property on type contexts in def. 18.

Definition 17 (TCP type-congruence). *The type congruence for the TCP-compliant subset of MAGπ, $\equiv_{\mathit{TCP}}$, is inductively defined using the same rules as $\equiv$ (fig. 4), but with the following change:*

$$\frac{}{\mathbb{M}_1 \cdot \mathbb{M}_2 \equiv \mathbb{M}_2 \cdot \mathbb{M}_1} \qquad \textit{replaced by} \qquad \frac{\mathsf{p} \neq \mathsf{q}}{\begin{array}{c} \mathsf{p}\,!\,\mathsf{m}_1(\mathbb{T}_1) \cdot \mathsf{q}\,!\,\mathsf{m}_2(\mathbb{T}_2) \cdot \mathbb{M} \\ \equiv_{\mathtt{TCP}} \mathsf{q}\,!\,\mathsf{m}_2(\mathbb{T}_2) \cdot \mathsf{p}\,!\,\mathsf{m}_1(\mathbb{T}_1) \cdot \mathbb{M} \end{array}}$$

Definition 18 (TCP safety). *Predicate $\varphi_{\mathtt{TCP}}$ is a Σ-**TCP-safety** property on typing contexts iff:*

$$\begin{array}{lll} \varphi_{\mathtt{TCP}}(\Gamma,\; s[\mathsf{p}] : \&_{i\in I}\{\mathsf{q}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\},\, s[\mathsf{q}]:\mathbb{M}) & & \\ \quad \text{and } \mathbb{M} \equiv_{\mathtt{TCP}} \mathsf{p}\,!\,\mathsf{m}(\mathbb{T}) \cdot \mathbb{M}' & & \\ \quad \text{and } \exists\, k \in I : \mathsf{q}_k = \mathsf{q} & \Longrightarrow & \mathsf{m}_k = \mathsf{m} \;\wedge\; \mathbb{T}_k = \mathbb{T} \\ \varphi_{\mathtt{TCP}}(\Gamma,\; s[\mathsf{p}] : \mu \mathrm{t}.\mathbb{S}) & \Longrightarrow & \varphi_{\mathtt{TCP}}(\Gamma,\; s[\mathsf{p}] : \mathbb{S}[^{\mu \mathrm{t}.\mathbb{S}}/_{\mathrm{t}}]) \\ \varphi_{\mathtt{TCP}}(\Gamma) \text{ and } \Gamma \rightarrow_{\Sigma} \Gamma' & \Longrightarrow & \varphi_{\mathtt{TCP}}(\Gamma') \end{array}$$

Similar to our previous definition of safety in def. 10, TCP safety ensures that payload types of communicating entities match. In addition, it also requires correct ordering of messages (up to $\equiv_{\mathtt{TCP}}$) by checking message labels—this is possible since messages between two parties do not get reordered, and so they must be received in the same order they are sent. In order to benefit from the session theorems proved in § 4, all that is required is to show that $\varphi_{\mathtt{TCP}} \subseteq \varphi_{\mathtt{s}}$, *i.e.*, any context that is TCP-safe is also safe. This is the only requirement since all theorems in § 4 (i) are parametric on the reliability function $\mathcal{R}$, including fully reliable networks; and (ii) are proven for $(\Sigma;\mathcal{R})$-$\varphi_{\mathtt{s}}(\Gamma)$.

Proposition 1 (Containment of $\varphi_{\mathtt{TCP}}$ in $\varphi_{\mathtt{s}}$). $\forall \Gamma \in \varphi_{\mathtt{TCP}} : \Gamma \in \varphi_{\mathtt{s}}$.

Proof. $\varphi_{\mathtt{TCP}}$ uses a fully reliable configuration of MAGπ—*i.e.,* is void of failure-handling timeouts—and thus trivially abides by [S-$\mathcal{R}_1$] and [S-$\mathcal{R}_2$]. [S-μ] is reflected directly in $\varphi_{\mathtt{TCP}}$. [S-$\rightarrow$] is reflected for $\mathcal{R} = \mathcal{R}_F$, *i.e.,* for a fully reliable configuration. [S-Com] is never violated by $\Gamma \in \varphi_{\mathtt{TCP}}$ since $\equiv_{\mathtt{TCP}} \subset \equiv$. $\square$

6 Case Study

This work presents the Ping (examples 1–5) and Domain Name System (§ 6.1) examples as they are widely known, and between them cover the full range of our contributions. Previous related works are *not* expressive enough to model either protocol with our range of failure assumptions. Thus Ping and DNS are suitable to illustrate how MAGπ pushes the boundaries of MPST. Additional examples are provided in the technical report [23].

6.1 DNS

We now demonstrate the key features of MAGπ through a case study. We present a multiparty example of a Domain Name System (DNS) with a cache and inbuilt load-balancer. This example: (i) reasons about failures in its unreliable connections that are specified using our novel *viewpoint-specific* reliability sets; (ii) defines *failure-handling* protocols for these possible failures; (iii) is *bounded* (def. 15), and thus has decidable type-level properties; and (iv) is *safe, $\mathcal{R}_F$-communication-safe, deadlock-free, terminating,* and *live.* Typically, DNS is implemented over TCP, however the distributed components can still suffer hardware failures. To cater for this, and for better demonstration of our contributions, we describe the protocol in our failure-prone setting.

Specification We consider a specification of a client-DNS interaction, where the client consults a cache, and the DNS delegates requests to workers.

The client, represented by role c, wishes to retrieve a web-address for a particular URL, and can do so by issuing a $\mathsf{request}$ to the DNS. As an optimisation, the client also stores recently retrieved addresses in a local and reliable cache—thus before issuing new requests to the DNS, it first consults this cache. Upon receiving a request, the DNS offloads processing work to one of two workers, represented by roles w_1 and w_2. After retrieving the appropriate address, the worker sends the response to the client.

The reliability configuration of this application is as such: the client and cache have reliable connections, formally $\mathcal{R}(\mathsf{c}) = \{\mathsf{cache}\}$ and $\mathcal{R}(\mathsf{cache}) = \{\mathsf{c}\}$; the DNS and workers have reliable connections, formally $\mathcal{R}(\mathsf{DNS}) = \{\mathsf{w}_1, \mathsf{w}_2\}$ and $\mathcal{R}(\mathsf{w}_1) = \mathcal{R}(\mathsf{w}_2) = \{\mathsf{DNS}\}$; all other communications are unreliable.

We now present the session types specifying the communication protocol for this distributed application. We adopt shorthand notion for singleton selections, and omit payload types for simplicity, as with the ping example.

Example 6 (DNS protocol).

$$\mathbb{S}_{\mathsf{c}} = \mathsf{cache}\,!\,\mathsf{req}().\,\&\begin{cases}\mathsf{cache}\,?\,\mathsf{ans}().\mathbf{end},\\ \mathsf{cache}\,?\,\mathsf{404}().\mathsf{DNS}\,!\,\mathsf{req}().\,\&\begin{cases}\mathsf{w}_1\,?\,\mathsf{ans}().\mathsf{cache}\,!\,\mathsf{new}().\mathbf{end},\\ \mathsf{w}_2\,?\,\mathsf{ans}().\mathsf{cache}\,!\,\mathsf{new}().\mathbf{end},\\ \text{⏲}.\,\mathsf{cache}\,!\,\mathsf{ko}().\mathbf{end}\end{cases}\end{cases}$$

$$\mathbb{S}_{\mathsf{cache}} = \&\begin{cases}\mathsf{c}\,?\,\mathsf{req}().\oplus\begin{cases}\mathsf{c}\,!\,\mathsf{ans}().\mathbf{end},\\ \mathsf{c}\,!\,\mathsf{404}().\,\&\begin{cases}\mathsf{c}\,?\,\mathsf{new}().\mathbf{end},\\ \mathsf{c}\,?\,\mathsf{ko}().\mathbf{end}\end{cases}\end{cases}\end{cases}$$

$$\mathbb{S}_{\mathsf{DNS}} = \&\begin{cases}\mathsf{c}\,?\,\mathsf{req}().\oplus\begin{cases}\mathsf{w}_1\,!\,\mathsf{req}().\mathsf{w}_2\,!\,\mathsf{ko}().\mathbf{end}\\ \mathsf{w}_2\,!\,\mathsf{req}().\mathsf{w}_1\,!\,\mathsf{ko}().\mathbf{end}\end{cases}\\ \text{⏲}.\,\mathsf{w}_1\,!\,\mathsf{ko}().\mathsf{w}_2\,!\,\mathsf{ko}().\mathbf{end}\end{cases}$$

$$\mathbb{S}_{\mathsf{w}_i} = \&\begin{cases}\mathsf{DNS}\,?\,\mathsf{req}().\mathsf{c}\,!\,\mathsf{ans}().\mathbf{end},\\ \mathsf{DNS}\,?\,\mathsf{ko}().\mathbf{end}\end{cases}$$

Our viewpoint-specific definition of reliability is necessary to specify the reliable connections with the DNS and workers whilst maintaining unreliable connections with the client. Additionally, the client type $\mathbb{S}_{\mathsf{c}}$ (resp. the DNS type $\mathbb{S}_{\mathsf{DNS}}$) is dependant on using undirected branching (resp. selection). Hence this example is not expressible using previous theory [4,33].

7 Related Work, Conclusions and Future Work

Modelling failures has become a relevant and widely researched topic in recent years. We elaborate on how our generic type system and modular language differs from, and in some cases may possibly subsume, related work.

Majumdar *et al.* [24] introduce undirected branching as a means of catering for the non-deterministic *partial* reordering of messages that is possible in networks using the Transmission Control Protocol (TCP). As shown in § 5, the modularity of our type system allows MAGπ to be adapted to support this network configuration, as well as other settings with lower levels of abstraction.

Affine type systems define types that can be used *at most once*. Affine session types [25,12,6] use affine typing metatheory to allow sessions to be prematurely cancelled in the event of failure. These works only model application-level failure (using try/catch blocks) and do not necessarily describe *how* a failure is handled, but only allow the initial protocol to be abandoned if failure occurs.

Viering *et al.* [38] present a MPST theory for event-driven distributed systems, where processes are restarted by monitors if they crash. This approach *requires* a centralised reliable node, a notion that is subsumed by our *view-point specific* definition of *reliability*, def. 7.

Chen *et al.* [8] remove the need for a centralised reliable node. They equip their type system with *synchronisation points* capable of detecting and handling failures raised by the nodes that experience them. Similarly, Adameit *et al.* [1] consider an environment free from a centralised reliable node where unstable *links* between participants can fail. They introduce the concept of *optional blocks*, allowing *default values* to substitute data not received due to communication failure. Viering *et al.* [37], motivated by consensus algorithms, delegate a group of processes as a permanently available recovery system capable of monitoring processes and informing them of failures. Thus, they no longer rely on *one* centralised robust node, but instead assume that at least some of the processes that make up the coordinator are alive at any given time. The drawback in these approaches is their reliance on coordination to handle faults. This may not be suitable with certain network configurations and failure-models. Since our type system handles failure through low-level techniques, it remains agnostic to the types of failures, and is suitable for any non-Byzantine network configuration.

Recent work by Peters *et al.* [28] extends global type theory with *failure annotations*—marking communication susceptible to failures and the kind of failure (specifically either process crashes or message loss). They handle failure by defining *default values and branches.* Since the theory is an extension of global types, it suffers from the same problems that are addressed through generalised MPST. Additionally, the work is not agnostic to failure-models, and so it is uncertain if the theory is capable of model failures other than the two considered.

Most similar to MAGπ is work by Barwell *et al.* [4], where generalised session type theory is extended to reason about crash-stop failures. They reserve the crash message label, which can be used in receive branches to detect node failure and specify failure-handling subprotocols. In line with our research, their type system is generic, thus improving its expressiveness. However, unlike MAGπ, their theory is not asynchronous, does not support undirected branching/selection, and assumes crash-stops to be the only possible faults—we address and capture a range of failures such as crash failures, link failures, message loss, delays and reordering and network partitioning.

Distributed variations of the π-calculus [2,30,7,13] introduce process *locations*—representations of real-world physical hardware. Processes are assigned to locations to form a topology, and locations can be crashed to model failures. None of these calculi model the range of failures that are supported by MAGπ, nor do they have type systems to ensure communication-safe failure handling.

To conclude the paper, we presented MAGπ—a Multiparty, Asynchronous and Generalised π-calculus which addresses the widest set of non-Byzantine faults by using timeouts and the most general reliability definition. Our language builds on the *generalised* and *asynchronous* MPST, which is the most flexible for distributed programming. We prove subject reduction and session fidelity; a series of process properties, as well as fault-handling safety and reliability adherence. As future work, we aim to investigate linear logic for Curry-Howard correspondences in order to understand the foundational and canonical meaning of faults and reliability. We aim to investigate Byzantine faults in combination with the non-Byzantine faults addressed here. Lastly, we will explore the use of model checking to streamline the verification of process properties.

Acknowledgements. We thank the anonymous reviewers and give a special thanks to Simon Fowler for his invaluable support and feedback.

References

1. Adameit, M., Peters, K., Nestmann, U.: Session types for link failures. In: Bouajjani, A., Silva, A. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10321, pp. 1–16. Springer (2017). https://doi.org/10.1007/978-3-319-60225-7_1
2. Amadio, R.M.: An asynchronous model of locality, failure and process mobility. In: Garlan, D., Métayer, D.L. (eds.) Coordination Languages and Models, Second International Conference, COORDINATION '97, Berlin, Germany, September 1-3, 1997, Proceedings. Lecture Notes in Computer Science, vol. 1282, pp. 374–391. Springer (1997). https://doi.org/10.1007/3-540-63383-9_92
3. Bartoletti, M., Scalas, A., Tuosto, E., Zunino, R.: Honesty by typing. Log. Methods Comput. Sci. **12**(4) (2016). https://doi.org/10.2168/LMCS-12(4:7)2016
4. Barwell, A.D., Scalas, A., Yoshida, N., Zhou, F.: Generalised multiparty session types with crash-stop failures. In: Klin, B., Lasota, S., Muscholl, A. (eds.) 33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland. LIPIcs, vol. 243, pp. 35:1–35:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.CONCUR.2022.35
5. Busi, N., Gabbrielli, M., Zavattaro, G.: On the expressive power of recursion, replication and iteration in process calculi. Math. Struct. Comput. Sci. **19**(6), 1191–1222 (2009). https://doi.org/10.1017/S096012950999017X
6. Capecchi, S., Giachino, E., Yoshida, N.: Global escape in multiparty sessions. Math. Struct. Comput. Sci. **26**(2), 156–205 (2016). https://doi.org/10.1017/S0960129514000164
7. Castellani, I.: Process algebras with localities. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 945–1045. North-Holland / Elsevier (2001). https://doi.org/10.1016/b978-044482830-9/50033-3
8. Chen, T., Viering, M., Bejleri, A., Ziarek, L., Eugster, P.: A type theory for robust failure handling in distributed systems. In: Albert, E., Lanese, I. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9688, pp. 96–113. Springer (2016). https://doi.org/10.1007/978-3-319-39570-8_7
9. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001)
10. Clarke, E.M., Klieber, W., Novácek, M., Zuliani, P.: Model checking and the state explosion problem. In: LASER Summer School. Lecture Notes in Computer Science, vol. 7682, pp. 1–30. Springer (2011). https://doi.org/10.1007/978-3-642-35746-6_1
11. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session Types for Object-Oriented Languages. In: ECOOP 2006. vol. 4067, pp. 328–352. Springer Berlin Heidelberg (2006). https://doi.org/10.1007/11785477_20, http://link.springer.com/10.1007/11785477_20, lecture Notes in Computer Science
12. Fowler, S., Lindley, S., Morris, J.G., Decova, S.: Exceptional asynchronous session types: session types without tiers. Proc. ACM Program. Lang. **3**(POPL), 28:1–28:29 (2019). https://doi.org/10.1145/3290341

13. Hennessy, M.: A distributed Pi-calculus. Cambridge University Press (2007)
14. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings. Lecture Notes in Computer Science, vol. 715, pp. 509–523. Springer (1993). https://doi.org/10.1007/3-540-57208-2_35
15. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP. LNCS, vol. 1381, pp. 122–138. Springer (1998). https://doi.org/10.1007/BFb0053567
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016). https://doi.org/10.1145/2827695
17. Kokke, W., Dardha, O.: Deadlock-free session types in linear haskell. In: Hage, J. (ed.) Haskell 2021: Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell, Virtual Event, Korea, August 26-27, 2021. pp. 1–13. ACM (2021). https://doi.org/10.1145/3471874.3472979
18. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with mungo and stmungo. In: Cheney, J., Vidal, G. (eds.) Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016. pp. 146–159. ACM (2016). https://doi.org/10.1145/2967973.2968595
19. Lagaillardie, N., Neykova, R., Yoshida, N.: Stay safe under panic: Affine rust programming with multiparty session types. In: 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany. LIPIcs, vol. 222, pp. 4:1–4:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.4
20. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998). https://doi.org/10.1145/279227.279229
21. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off go: liveness and safety for channel-based programming. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 748–761. ACM (2017). https://doi.org/10.1145/3009837.3009847
22. Laprie, J.C.: Dependable computing and fault-tolerance. Digest of Papers FTCS-15 **10**(2), 124 (1985)
23. Le Brun, M.A., Dardha, O.: Magπ: Types for failure-prone communication (2023). https://doi.org/10.48550/ARXIV.2301.10827, https://arxiv.org/abs/2301.10827
24. Majumdar, R., Mukund, M., Stutz, F., Zufferey, D.: Generalising projection in asynchronous multiparty session types. In: Haddad, S., Varacca, D. (eds.) 32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference. LIPIcs, vol. 203, pp. 35:1–35:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.35
25. Mostrous, D., Vasconcelos, V.T.: Affine sessions. Log. Methods Comput. Sci. **14**(4) (2018). https://doi.org/10.23638/LMCS-14(4:14)2018
26. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: USENIX Annual Technical Conference. pp. 305–319. USENIX Association (2014), https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro
27. Orchard, D., Yoshida, N.: Session types with linearity in haskell. Behavioural Types: from Theory to Tools p. 219 (2017)

28. Peters, K., Nestmann, U., Wagner, C.: Fault-tolerant multiparty session types. In: Mousavi, M.R., Philippou, A. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 42nd IFIP WG 6.1 International Conference, FORTE 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13273, pp. 93–113. Springer (2022). https://doi.org/10.1007/978-3-031-08679-3_7
29. Pierce, B.C.: Types and programming languages. MIT Press (2002)
30. Riely, J., Hennessy, M.: Distributed processes and location failures. Theor. Comput. Sci. **266**(1-2), 693–735 (2001). https://doi.org/10.1016/S0304-3975(00)00326-1
31. Rossi, M.: Modeling and analysis of communicating systems. Formal Aspects Comput. **33**(2), 297–298 (2021). https://doi.org/10.1007/s00165-021-00533-8
32. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: Müller, P. (ed.) 31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain. LIPIcs, vol. 74, pp. 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.24
33. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Proc. ACM Program. Lang. **3**(POPL), 30:1–30:29 (2019). https://doi.org/10.1145/3290343
34. Tabone, G., Francalanza, A.: Session types in elixir. In: Castegren, E., Koster, J.D., Fowler, S. (eds.) AGERE 2021: Proceedings of the 11th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, Virtual Event / Chicago, IL, USA, 17 October 2021. pp. 12–23. ACM (2021). https://doi.org/10.1145/3486601.3486708
35. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: PARLE '94. LNCS, vol. 817, pp. 398–413. Springer (1994)
36. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012). https://doi.org/10.1016/j.ic.2012.05.002
37. Viering, M., Chen, T., Eugster, P., Hu, R., Ziarek, L.: A typing discipline for statically verified crash failure handling in distributed systems. In: Ahmed, A. (ed.) Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10801, pp. 799–826. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_28
38. Viering, M., Hu, R., Eugster, P., Ziarek, L.: A multiparty session typing discipline for fault-tolerant event-driven distributed programming. Proc. ACM Program. Lang. **5**(OOPSLA), 1–30 (2021). https://doi.org/10.1145/3485501

System F_ω^μ with Context-free Session Types[⋆]

Diogo Poças(✉), Diana Costa, and Andreia Mordido, and Vasco T. Vasconcelos

LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal
{dmpocas,dfdcosta,afmordido,vmvasconcelos}@ciencias.ulisboa.pt

Abstract. We study increasingly expressive type systems, from F^μ—an extension of the polymorphic lambda calculus with equirecursive types—to $F_\omega^{\mu;}$—the higher-order polymorphic lambda calculus with equirecursive types and context-free session types. Type equivalence is given by a standard bisimulation defined over a novel labelled transition system for types. Our system subsumes the contractive fragment of F_ω^μ as studied in the literature. Decidability results for type equivalence of the various type languages are obtained from the translation of types into objects of an appropriate computational model: finite-state automata, simple grammars and deterministic pushdown automata. We show that type equivalence is decidable for a significant fragment of the type language. We further propose a message-passing, concurrent functional language equipped with the expressive type language and show that it enjoys preservation and absence of runtime errors for typable processes.

Keywords: System F, Higher-order kinds, Context-free session types

1 Introduction

Extensions of the λ-calculus to include increasingly sophisticated type structures have been extensively studied and have led to systems whose importance is widely recognized: System F [60], System F^μ [30], System F_ω [36], System F_ω^μ [14]. Ideally, we would like to combine a *wishlist* of type structures and get a super-powerful system with vast expressiveness. However, the expressiveness of types is naturally limited by the universe where they are supposed to live: programming languages. Expressive type systems pose challenges to compilers that other (less expressive) types do not even reveal; one such example is type equivalence checking.

System F can be enriched with different type constructors for specifying communication protocols. We analyse the impact of combinations of such constructors on the type equivalence problem. In order to do so, we extend System F with session types [42,43,67]. Session types provide for detailed protocol specifications in the form of types. Traditional recursive session types are limited to tail

[⋆] Support for this research was provided by the Fundação para a Ciência e a Tecnologia through project SafeSessions, ref. PTDC/CCI-COM/6453/2020, and by the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020. A full version is available on arXiv [20].

recursion, thus failing to capture all protocols whose traces cannot be characterized by regular languages. Context-free session types overcome this limitation by extending types with a notion of sequential composition, $T;U$ [2,68]. The set of types together with the $;$ binary operation constitutes a monoid, for which a new type, Skip, acts as the neutral element and End acts as an absorbing element.

The regular recursive type $\mu\,\alpha\colon \mathrm{S}.\,\&\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon ?\mathsf{Int};\alpha\}$ describes an integer *stream* as seen from the point of view of the consumer. It offers a choice between Done—after which the channel must be closed (as witnessed by type End)—and More—after which an integer value must be received, followed by the rest of the stream. Types are categorised by *kinds*, so that we know that the recursion variable α is of kind session—denoted by S—and, thus, can be used with semicolon. Instead, we might want to write a type with a more *context-free* flavour. The type $\mu\,\alpha\colon \mathrm{S}.\,\&\{\mathsf{Leaf} : \mathsf{Skip}, \mathsf{Node} : \alpha; ?\mathsf{Int};\alpha\};\mathsf{End}$ describes a protocol for the type-safe streaming of integer *trees* on channels. The continuation to the Leaf option is Skip, where no communication occurs but the channel is still open for further composition. The continuation to the Node choice receives a left subtree, an integer at the root and a right subtree. In either case, once the whole tree is received, the channel must be closed, as witnessed by the final End. Beyond first-order context-free session types (where only basic types are exchanged) [2,68] we may be interested in higher-order session types capable of exchanging values of complex types [19]. A goal of this paper is the integration of higher-order context-free session types into system F^{μ}_{ω}. We want to be able to abstract the type that is received on a tree channel, which is now possible by writing $\lambda\alpha\colon \mathrm{T}.\mu\,\beta\colon \mathrm{S}.\,\&\{\mathsf{Leaf} : \mathsf{Skip}, \mathsf{Node} : \beta; ?\alpha; \beta\};\mathsf{End}$, where T is the kind of functional types.

A form of abstraction over session types with general recursion was proposed by Das et al. [24,25] via (nested) parametric polymorphism. In the notation of Das et al., we can write a type equation for abstracting the type being received on a stream channel $\mathsf{Stream}\langle\alpha\rangle \doteq \&\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon ?\alpha; \mathsf{Stream}\langle\alpha\rangle\}$. Using abstractions, we can write Stream as a function of its parameter α, $\mathsf{Stream} \doteq \lambda\alpha\colon \mathrm{T}.\&\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon ?\alpha; \mathsf{Stream}\,\alpha\}$; alternatively, we can use the μ-operator to rewrite the Stream type as $\lambda\alpha\colon \mathrm{T}.(\mu\,\beta\colon \mathrm{S}.\,\&\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon ?\alpha.\beta\})$. Das et al. proved that parametrized type definitions over regular session types are strictly more expressive than context-free session types. To some extent, this analogy guides our approach: if adding abstraction (via parametric polymorphism) to regular types leads to nested types, what exactly does it mean to add abstraction (via a type-level λ-operator) to context-free types? Throughout this paper we analyse several increments to System F^{μ} that culminate in adding λ-abstraction to context-free session types.

One of our focuses is necessarily the analysis of the type equivalence problem. The uncertainty about the decidability of this problem over recursive parametric types goes back to the 1970s [16,63]. Although the type equivalence problem for parametric (nested) session types and context-free session types is decidable, that for the combination of abstractions over context-free types may no longer be. In fact, this analysis constitutes an interesting journey towards a better understanding of the role of higher-order polymorphic recursion in presence of sequential composition, as well as the gains (and losses) resulting from combining abstraction with arbitrary (rather than tail) recursion.

Ultimately, decidability is not a sufficiently valuable measure regarding a type system's *practicality*. We look for type systems that may be incorporated into compilers. For that reason, we are interested in algorithms for type equivalence checking. Equivalence in F_ω^μ alone is already at least as hard as equivalence of deterministic pushdown automata. If we restrict recursion to the monomorphic case (requiring recursion variables to denote proper types, that is of kind S or T, collectively denoted by $*$) we lower the complexity of type equivalence to that of equivalence for finite-state automata. The extension with context-free session types is slightly more complex. In order to obtain "good" algorithms, we restrict the recursion to the monomorphic case, arriving at classes $F_\omega^{\mu *}, F_\omega^{\mu *;}$. Now the type equality problem for $F_\omega^{\mu *;}$ translates to the equivalence problem for simple grammars, which is still decidable [4,33]. Since $F_\omega^{\mu *;}$ subsumes $F_\omega^{\mu *}$, our proof of the decidability of type equivalence serves as an alternative to that of Cai et al. [14] (restricted to contractive types).

Higher-order polymorphism allows for the definition of type operators and the internalisation of various (session-type) constructs that would otherwise be offered as built-in constructors. In this way, we are able to internalise basic session-type constructors such as sequential composition $;$ and the Dual type operator (which reverses the direction of communication between parties). Duality is often treated as an external macro. Gay et al. [34] explore different ways of handling the dual operator, all in a monomorphic setting. In the presence of polymorphism the dual operator cannot be fully eliminated without introducing co-variables. Internalisation offers a much cleaner solution.

Due to the presence of sequential composition, regular trees are *not* a powerful enough model for representing types (`type TreeC a` in Section 2 is an example). The main technical challenge when combining System F_ω^μ and context-free session types is making sure that the resulting model can still be represented by simple grammars, so that type equivalence may be decided by a practical algorithm. The difficulties arise with renaming bound variables. For infinite types, both renaming with fresh variables and using de Bruijn indices may create an infinite number of distinct variables, which makes the construction of a simple grammar simply impossible. For example, take the type $\lambda\alpha\colon \mathrm{T}.\mu\gamma\colon \mathrm{T}.\lambda\beta\colon \mathrm{T}.\alpha \to \gamma$, which stands for the infinite type $\lambda\alpha\colon \mathrm{T}.\lambda\beta\colon \mathrm{T}.\alpha \to \lambda\beta\colon \mathrm{T}.\alpha \to \lambda\beta\colon \mathrm{T}\ldots$ Renaming this type using a fresh variable at each step would result in a type of the form $\lambda v_1\colon \mathrm{T}.\lambda v_2\colon \mathrm{T}.v_1 \to \lambda v_3\colon \mathrm{T}.v_1 \to \lambda v_4\colon \mathrm{T}\ldots$, requiring infinitely many variables. Similarly, de Bruijn indices [27] yield a type of the form $\lambda_{\mathrm{T}}\lambda_{\mathrm{T}}1 \to \lambda_{\mathrm{T}}2 \to \lambda_{\mathrm{T}}3 \to \ldots$ that requires an infinite number of natural indices. We thus introduce *minimal renaming* that uses the least amount of variable names as possible (cf. Gauthier and Pottier [30]). This ensures that only finitely many terminal symbols are necessary, allowing for translating types into simple grammars.

Type languages live in term languages and we propose a term language to consume $F_\omega^{\mu;}$ types. Based on Almeida et al. [2], we introduce a message-passing concurrent programming language. Type checking is decidable if type equivalence is, and it is, in particular, for $F^{\mu *;}_{\omega}$.

The main contributions of this paper are as follows.

– The integration of (higher-order) context-free session types into system F^{μ}_{ω}, dubbed $F^{\mu;}_{\omega}$.
– A semantic definition of type equivalence via a labelled transition system.
– The identification of a suitable fragment of System $F^{\mu;}_{\omega}$ for which type equivalence is reduced to the bisimilarity of simple grammars.
– A proof that type equivalence on the full System $F^{\mu;}_{\omega}$ is at least as hard as bisimilarity of deterministic pushdown automata.
– The first internalisation of the Dual type operator in a type language.
– A term language to consume $F^{\mu;}_{\omega}$ types and an accompanying metatheory.

The type system presented in the paper combines three constructions: sequential composition of session types, higher-order kinds via type-level abstraction and application, and higher-order recursion. Prior to our work there is the system by Almeida et al. [4] which incorporates sequential composition and (first-order) recursion, but no higher-order kinds. There is also the system by Cai et al. [14] which incorporates higher-order kinds and higher-order recursion, but no sequential composition. Our system is the first to incorporate all three constructions. Although some of the results are incremental and generalize results from the literature, the main technical challenge is understanding the border past which they do not hold anymore. For example, "just" including higher-order kinds into the system by Almeida et al. does not work, since we need to pay close attention to variable names, making sure that type equivalence is invariant with respect to alpha-conversion (renaming of bound variables). This called for a novel notion of renaming, inspired by Gauthier and Pottier [30]. Similarly, "just" including sequential composition into the system of Cai et al. does not work, since finite-state automata (or regular trees) are not enough to capture the expressive power of the new type system, *even* when restricted to first-order recursion. This required us to look at the more expressive framework of simple grammars, and introduce a translation from types to words of a simple grammar.

The rest of the paper is organised as follows. The next section motivates the type language and introduces the term language with an example. Section 3 introduces System $F^{\mu;}_{\omega}$, Section 4 discusses type equivalence and Section 5 shows that type equivalence is decidable for a fragment of the type language. Section 6 presents the term language and its metatheory. Section 7 discusses related work and Section 8 concludes the paper with pointers for future work. Proofs for the main results can be found in a technical report on arXiv [20].

2 Motivation

Our goal is to study type systems that combine equirecursion, higher-order polymorphism, and higher-order context-free session types, while incorporating these in programming languages.

$$(\!|\,|\!) ::= \{\} \mid \langle\rangle \qquad \sharp ::= ? \mid ! \qquad \odot ::= \& \mid \oplus \qquad * ::= \textsc{t} \mid \textsc{s}$$

$$\begin{array}{llll}
T & ::= T \rightarrow T \mid (\!|\overline{l_i\colon T_i}|\!) \mid \forall\alpha\colon\kappa.T \mid \mu\,\alpha\colon\kappa.T \mid \alpha & (F^{\mu}) & \kappa = \textsc{t} \\
T & ::= (F^{\mu}) \mid \sharp T.T \mid \odot\{\overline{l_i\colon T_i}\} \mid \mathsf{End} & (F^{\mu\cdot}) & \kappa = * \\
T & ::= (F^{\mu}) \mid \sharp T \mid \odot\{\overline{l_i\colon T_i}\} \mid \mathsf{End} \mid T;T \mid \mathsf{Skip} & (F^{\mu;}) & \kappa = * \\
T & ::= (F^{M}) \mid \lambda\alpha\colon\kappa.T \mid T\,T & (F^{M}_{\omega}), M ::= \mu, \mu\cdot, \mu; & \\
 & & & \kappa = * \mid \kappa \Rightarrow \kappa
\end{array}$$

Fig. 1: Six F-systems.

Extensions of System F. Figure 1 motivates the construction by proposing six different type languages, culminating with $F^{\mu;}_{\omega}$. The initial system, F^{μ}, includes well-known basic type operators [57]: functions $T \rightarrow U$, records $\{\overline{l_i\colon T_i}\}$ and variants $\langle\overline{l_i\colon T_i}\rangle$. Type Unit is short for $\{\}$, the empty record; we can imagine that Unit stands in place of an arbitrary scalar type such as Int and Bool. We also include variable names α, type quantification $\forall\alpha\colon\kappa.T$ and recursion $\mu\,\alpha\colon\kappa.T$. To control type formation, all variable bindings must be kinded with some kind κ, even if for the initial system, F^{μ}, we only use the functional kind $\textsc{t}$.

We then build on F^{μ} by considering (regular, tail recursive) session types; we represent the resulting system by $F^{\mu\cdot}$. For example $?\mathsf{Int}.!\mathsf{Bool}.\mathsf{End}$ is a type for a channel endpoint that receives an integer, sends a boolean, and terminates. At this point we introduce a kind $\textsc{s}$ of session types to restrict the ways in which we can combine session and functional types together. For example, a well-formed type $?T.U$ is of kind $\textsc{s}$ and requires U to be also of kind $\textsc{s}$ (whereas T can be of kind $*$, that is $\textsc{s}$ or $\textsc{t}$). An example of an infinite session type is $\mu\,\alpha\colon\textsc{s}.\,!\mathsf{Int}.\alpha$ that endlessly outputs integer values. For a more elaborate example consider the type $\mathsf{IntStream} = \mu\,\alpha\colon\textsc{s}.\,\&\{\mathsf{Done}\colon\mathsf{End}, \mathsf{More}\colon ?\mathsf{Int}.\alpha\}$ that specifies a channel endpoint for receiving a (finite or infinite) stream of integer values. Communication ends after choice Done is selected.

The next step of our construction takes us to context-free session types; the resulting system is denoted by $F^{\mu;}$. We introduce a new construct for sequential composition $T;U$, and a new type Skip, acting as the neutral element of sequential composition [68]. The message constructors are now unary ($?T$ and $!T$) rather than binary. In System $F^{\mu;}$ we distinguish between the traditional End type and the Skip type. These types have different behaviours: End terminates a channel, while Skip allows for further communication. Type equality is more subtle for context-free session types, because of the monoidal semantics of sequential composition. It is derivable from the following axioms:

$$\begin{array}{rcll}
\mathsf{Skip};T & \sim & T & \text{Neutral element} \\
\mathsf{End};T & \sim & \mathsf{End} & \text{Absorbing element} \\
(T;U);V & \sim & T;(U;V) & \text{Associativity} \\
\odot\{\overline{l_i\colon T_i}\};U & \sim & \odot\{\overline{l_i\colon T_i;U}\} & \text{Distributivity}
\end{array} \qquad (1)$$

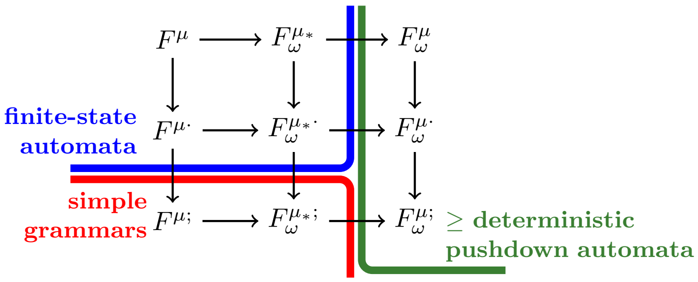


Fig. 2: Relation between the main classes of types in this paper (arrows denote strict inclusions).

Although the syntax of $F^{\mu\cdot}$ is not formally included in the syntax of $F^{\mu;}$, we can embed recursive session types into context-free session types by mapping $\sharp T.U$ into $\sharp T;U$. It is well-known that context-free session types allow for higher computational expressivity: while F^{μ} and $F^{\mu\cdot}$ can be represented via finite-state automata, $F^{\mu;}$ can only be represented with simple grammars [4,33].

To finalise our construction, we include type abstraction $\lambda\alpha\colon \kappa.T$ and type application $T\ U$. Again, type abstraction binds a variable which must be kinded. Kinds can now be of higher-order $\kappa \Rightarrow \kappa'$. For each of the three systems F^{μ}, $F^{\mu\cdot}$, $F^{\mu;}$ we arrive at a higher-order version, respectively F^{μ}_{ω}, $F^{\mu\cdot}_{\omega}$, $F^{\mu;}_{\omega}$ (all of which we represent as F^{M}_{ω}). In System $F^{\mu\cdot}_{\omega}$, for example, we can specify channels for receiving (finite or infinite) sequences of values of arbitrary (but fixed) types,

$$\mathsf{Stream} = \lambda\alpha\colon \textsc{t}.(\mu\,\beta\colon \textsc{s}.\,\&\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon\, ?\alpha.\beta\})$$

where α can be instantiated with the desired type; in particular, $\mathsf{Stream\ Int}$ would be equivalent to the aforementioned $\mathsf{IntStream}$.

It turns out that the expressive power of general higher-order systems F^{M}_{ω} is too large for practical purposes. Even the simplest case F^{μ}_{ω} is at least as expressive as deterministic pushdown automata (or equivalently, first-order grammars), for which known equivalence algorithms are notoriously impractical. By impractical we mean that, although there exists a proof of decidability (due to Sénizergues [61], later improved by Stirling and Jancar [46,65]), the underlying algorithm is rather complex. To the best of our knowledge, there is no practical implementation of an algorithm to decide the equivalence of deterministic pushdown automata. This is essentially due to polymorphic recursion, which can be encoded by a higher-order μ-operator (we provide an example at the end of Section 5). Therefore, it makes sense to restrict the kind κ of the recursion operator $\mu\,\alpha\colon \kappa.\,T$. We use the notation μ_* to mean the subclass of types written using only $*$-kinded recursion, i.e., $\mu\,\alpha\colon \textsc{t}.\,T$ or $\mu\,\alpha\colon \textsc{s}.\,T$.

Figure 2 summarizes the main relations between the classes of types in our paper. Firstly, we obtain a lattice where the expressive power increases as we travel down (from functional to session to context-free session types) and right (from simple polymorphism to higher-order polymorphism with monomorphic recursion to arbitrary recursion). Four of the classes can be represented using finite-state automata (up to $F_\omega^{\mu_*\cdot}$). By including sequential composition ($F^{\mu;}$ and $F_\omega^{\mu_*;}$) we are still able to represent types using simple grammars. Once we allow for arbitrary recursion, the expressiveness of our model requires the computational power of deterministic pushdown automata.

Programming with $F_\omega^{\mu;}$*.* We now turn our attention to the term language, a message passing, concurrent functional language, equipped with context-free session types. Start with a stream of values of type `a`. Such a stream, when seen from the side of the reader, offers two choices: `Done` and `More`. In the former case the interaction is over; in the latter the reader reads a value of type `a`, as in `?a`, and recurses. This is the stream type we have seen before only that, rather than closing the channel endpoint (with type `End`), it terminates with type `Skip`, so that it may be sequentially composed with other types. In this informal introduction to the term language we omit the kinds of type variables.

```
type Stream a = &{Done: Skip, More: ?a ; Stream a}
```

A fold channel, as seen from the side of the folder, is a type of the following form. We assume that application binds tighter than semicolon, that is, type `Stream a ; !b ; End` is interpreted as `(Stream a) ; !b ; End`.

```
type Fold a b = ?(b → a → b) ; ?b ; Stream a ; !b ; End
```

Consumers of this type first receive the folding function, then the starting element, then the elements to fold in the form of a stream, and finally output the result of the fold. The type terminates with `End` for we do not expect type `Fold` to be further composed. Compare `Fold` with the type for a conventional functional left fold: `(b → a → b) → b → List a → b`.

We now develop a function that consumes a `Fold` channel. Syntax `x ▷ f` is for the inverse function application with low priority, that is `x ▷ f ▷ g = g (f x)`. Recall that `Unit` is an alternative notation for the empty record type, `{}`.

```
foldServer : ∀a.∀b. Fold a b → Unit
foldServer c = let (f, c) = receive c in
               let (e, c) = receive c in foldS f e c

foldS : ∀a.∀b. (b → a → b) → b → Stream a;!b;End → Unit
foldS f e c = match c with
  { Done c → c ▷ send e ▷ close
  , More c → let (x, c) = receive c in foldS f (f e x) c
  }
```

Function `foldServer` consumes the initial part of the channel and passes the rest of the channel to the recursive function `foldS` that consumes the whole stream while accumulating the fold value. In the end, when branch `Done` is selected, the fold value is written on the channel and the channel closed. In general, the channel operators—`receive`, `send`, `select`—return the same channel in the form of a new identifier. It is customary to reuse the identifier name—`c` in the example, as in `let (f, c)= receive c`—since it denotes the same channel. Syntax `c ▷ ...`

hides the continuation channel. The case for the external choice—`match`—also returns the continuation (in each branch) so that interaction on the channel endpoint may proceed.

We may now write different clients for the `foldServer`. Examples include a client that generates a stream from a pair of integer values (denoting an interval); another that generates the stream from a list of values; and yet another that generates the stream from a binary tree. We propose a further client. Consider the type of a channel that exchanges trees in a serialized format [68]. Its polymorphic version, as seen from the point of view of the reader, is as follows:

```
type TreeChannel a = TreeC a ; End
type TreeC a = &{Leaf: Skip, Node: TreeC a;?a;TreeC a}
```

We transform trees as we read from tree channels into streams. Function `flatten` receives a tree channel and a stream channel (as seen from the point of view of the writer, hence the `Dual`) and returns the unused part of the latter.

```
flatten : ∀a.∀c. TreeChannel a → (Dual Stream a);c → c
```

We are now in a position to write a client that checks whether all values in a tree channel are positive.

```
allPositive : TreeChannel Int → Dual (Fold Int Bool) → Bool
allPositive t c =
  let c = send (λx:Bool.λy:Int. x && y > 0) c in
  let c = send True c in
  let c = flatten [Int] [?Bool;End] t c in
  let (x, c) = receive c in
  close c; x
```

The client sends a function and the starting value on the fold channel. Then, it flattens the given tree `t`, receives the folded value and closes the channel. Syntax `flatten [Int] [?Bool;End]` is for term-level type application. We mean to flatten a tree of `Int` values on a stream channel whose continuation is of type `?Bool;End`. The continuation channel is bound to `c` so that we may further receive the fold value and thereupon close the channel. Syntax `e1;e2` is for sequential composition and abbreviates `let {} = e1 in e2` given that `{}`, the `Unit` value, is linear and hence must be consumed.

Finally, a simple application creates a new `TreeC` channel, passing one end to a thread that produces a tree channel. Function `new` creates a channel and returns its two ends. It then creates a `Fold` channel, distributes one end to a thread `foldServer` and the other to function `allPositive`. The `fork` primitive receives a suspended computation (a thunk, of the form `λx:Unit.e`) and creates a new thread that runs in parallel with that from where the `fork` was issued.

```
system : Bool
system = let (tr, tw) = new [TreeC Int] () in
  fork (λ_:Unit. produce tw);
  let (fr, fw) = new [Fold Int Bool] () in
  fork (λ_:Unit. foldServer fr);
  allPositive tr fw
```

$* ::=$		Kind of proper types
	S	session
	T	functional
$\kappa ::=$		Kind
	$*$	kind of proper types
	$\kappa \Rightarrow \kappa$	kind of type operators
$T ::=$		Type
	ι	type constant
	α	type variable
	$\lambda\alpha\colon \kappa.T$	type-level abstraction
	$T\,T$	type-level application

Fig. 3: The syntax of types.

$\iota ::=$		Type constant
$\rightarrow$	$* \Rightarrow * \Rightarrow \mathrm{T}$	arrow
$(\!\lvert \overline{l_i} \rvert\!)$	$\overline{* \Rightarrow} \mathrm{T}$	record, variant
μ_κ	$(\kappa \Rightarrow \kappa) \Rightarrow \kappa$	recursive type
$\forall_\kappa$	$(\kappa \Rightarrow *) \Rightarrow \mathrm{T}$	universal type
Skip	S	skip
End	S	end
$\sharp$	$* \Rightarrow \mathrm{S}$	input, output
$;$	$\mathrm{S} \Rightarrow \mathrm{S} \Rightarrow \mathrm{S}$	seq. composition
$\odot\{\overline{l_i}\}$	$\overline{\mathrm{S} \Rightarrow} \mathrm{S}$	choice operators
Dual	$\mathrm{S} \Rightarrow \mathrm{S}$	dual operator

Fig. 4: Type constants and kinds.

Type renaming $\boxed{\mathrm{rename}_S(T)}$

$$\mathrm{rename}_S(\iota) = \iota \quad \mathrm{rename}_S(\alpha) = \alpha \quad \mathrm{rename}_S(T\,U) = \mathrm{rename}_{S\cup\mathrm{fv}(U)}(T)\,\mathrm{rename}_S(U)$$

$$\mathrm{rename}_S(\lambda\alpha\colon \kappa.T) = \lambda\upsilon\colon \kappa.\mathrm{rename}_S(T[\upsilon/\alpha]) \quad \text{where } \upsilon = \mathrm{first}_S(\lambda\alpha\colon \kappa.T)$$

Fig. 5: Type renaming.

3 Kinds and Types

This section introduces in detail System $F_\omega^{\mu;}$, an extension of System F_ω^{μ} incorporating higher-order context-free session types. The syntax of types is presented in Fig. 3. A type is either a constant ι (as in Fig. 4), a type variable α, an abstraction $\lambda\alpha\colon \kappa.T$ or an application $T\,U$. Besides incorporating the standard session type constructors as constants, system $F_\omega^{\mu;}$ also includes Dual as a constant for a type operator mapping a session type to its dual. Note also that $\forall\alpha\colon \kappa.\,T$ is syntactic sugar for $\forall_\kappa(\lambda\alpha\colon \kappa.T)$. Analogously, $\mu\,\alpha\colon \kappa.\,T$ abbreviates $\mu_\kappa(\lambda\alpha\colon \kappa.T)$. This simplifies our analysis as lambda abstraction becomes the only binding operator.

A distinction between session and functional types is made resorting to kinds S and T, respectively. These are the kinds of proper types, $*$; we use the symbol κ to represent either the kind of a proper type or that of a type operator, of the form $\kappa \Rightarrow \kappa'$. A kinding context Δ stores kinds for type variables using bindings of the form $\alpha\colon \kappa$. Notation $\Delta + \alpha\colon \kappa$ denotes the update of kinding context Δ, defined as $(\Delta, \alpha\colon \kappa) + \alpha\colon \kappa' = \Delta, \alpha\colon \kappa'$ and $\Delta + \alpha\colon \kappa = \Delta, \alpha\colon \kappa$ when $\alpha \notin \Delta$.

To define type formation, we require a few notions. Firstly comes the notion of *renaming*, adapted from Gauthier and Pottier [30] and presented in Fig. 5. Renaming essentially replaces a type T by a minimal alpha-conversion of T. By alpha-conversion we mean that $\text{rename}_S(T)$ renames bound variables in T. By "minimal" we mean that each bound variable is renamed to its lowest possible value. We assume at our disposal a countable well-ordered set of type variables $\{\upsilon_1, \ldots, \upsilon_n, \ldots\}$. In $\text{rename}_S(T)$, parameter S is a set containing type variables unavailable for renaming; in the outset of the renaming process S is the empty set, since all variables are available. In that case the subscript S is often omitted. The case for lambda abstraction renames the bound variable by the smallest variable not in the set $S \cup \text{fv}(\lambda\alpha\colon \kappa.T)$, which we denote by $\text{first}_S(\lambda\alpha\colon \kappa.T)$.

Renaming is what allows us to check whether type abstractions $\lambda\alpha\colon \kappa.T$, $\lambda\beta\colon \kappa.U$ are equivalent. For the types to be equivalent, both bound variables α and β ought to be renamed to the same variable υ_j. In summary, renaming provides a syntax-guided approach to the equivalence of lambda-abstractions, where the names of bound variables should not matter. Our notion of type equivalence preserves alpha-conversions up to renaming: if T and U only differ on bound variables, then $\text{rename}(T) = \text{rename}(U)$ and in particular $\text{rename}(T) \sim \text{rename}(U)$. We will come back to this point after we define type equivalence in Section 4.

We can easily see that renaming uses the minimum amount of variable names possible; for example, $\text{rename}(\lambda\alpha\colon \text{T}.\lambda\beta\colon \text{S}.\beta) = \lambda\upsilon_1\colon \text{T}.\lambda\upsilon_1\colon \text{S}.\upsilon_1$. Notice how both bound variables α and β are renamed to υ_1, the first variable available for replacement. Also, renaming blatantly violates the Barendregt's variable convention [9] used in so many works; for example $\text{rename}(\upsilon_1\,(\lambda\alpha\colon T.\alpha)) = \upsilon_1\,(\lambda\upsilon_1\colon T.\upsilon_1)$, where variable υ_1 is both free and bound in the resulting type. Even if renaming violates the variable convention, substitution can still be performed without resorting to the "on-the-fly" renaming of Curry and Feys [21,40]. When $\upsilon_1 \neq \upsilon_2$, we have that

$$(\lambda\upsilon_1\colon \kappa.\lambda\upsilon_2\colon \kappa'.U)\,T \quad \text{reduces to} \quad \text{rename}((\lambda\upsilon_2\colon \kappa'.U)[T/\upsilon_1]).$$

Then, we have $(\lambda\upsilon_2\colon \kappa'.U)[T/\upsilon_1] = \lambda\upsilon_2\colon \kappa'.(U[T/\upsilon_1])$ since the renaming rule for application guarantees that $\upsilon_2 \notin \text{fv}(T)$. Otherwise if $\upsilon_1 = \upsilon_2$, we have $(\lambda\upsilon_1\colon \kappa'.U)[T/\upsilon_1] = \lambda\upsilon_1\colon \kappa'.U$. This justifies the inclusion of set S in the renaming process. From now on, we assume that all types have gone through the renaming process.

Next comes the notion of *type reduction* (Fig. 6). Apart from beta reduction (rule R-β), the definition provides for sequential composition, for unfolding recursive types and for reducing $\mathsf{Dual}\,T$ types. Note that renaming is further invoked in rule R-β for beta reduction does not preserve renaming: consider the renamed type $(\lambda\upsilon_1\colon \text{T}.\lambda\upsilon_2\colon \text{T}.\upsilon_1 \to \upsilon_2)\,\mathsf{Unit}$. The type resulting from the substitution $(\lambda\upsilon_2\colon \text{T}.\upsilon_1 \to \upsilon_2)[\mathsf{Unit}/\upsilon_1]$ is $\lambda\upsilon_2\colon \text{T}.\mathsf{Unit} \to \upsilon_2$ which is not renamed and, therefore, not equivalent to $\lambda\upsilon_1\colon \text{T}.\mathsf{Unit} \to \upsilon_1$ according to our rules in Section 4. Thanks to our modified rule R-β, we preserve renaming under reductions: if $T = \text{rename}(T)$ and $T \longrightarrow U$ then $U = \text{rename}(U)$.

We also need the notion of *weak head normal form* borrowed from the lambda calculus [9,10]. We say that a type T is in weak head normal form, $T\,\mathsf{whnf}$, if it is irreducible, i.e., $T \not\longrightarrow$. Although this is a negative definition, in the technical report we provide an equivalent, rule-based characterisation of weak head normal

Type reduction $\boxed{T \longrightarrow T}$

$$\textsc{R-Seq1}\quad \mathsf{Skip}; T \longrightarrow T \qquad \textsc{R-Seq2}\quad \frac{T \longrightarrow V}{T;U \longrightarrow V;U} \qquad \textsc{R-Assoc}\quad (T;U);V \longrightarrow T;(U;V) \qquad \textsc{R-}\mu\quad \mu_k\, T \longrightarrow T\,(\mu_k\, T)$$

$$\textsc{R-}\beta\quad (\lambda\alpha\colon \kappa.T)\, U \longrightarrow \mathrm{rename}(T[U/\alpha]) \qquad \textsc{R-TAppL}\quad \frac{T \longrightarrow U}{TV \longrightarrow UV} \qquad \textsc{R-D;}\quad \mathsf{Dual}\,(T;U) \longrightarrow \mathsf{Dual}\,T; \mathsf{Dual}\,U$$

$$\textsc{R-DSkip}\quad \mathsf{Dual}\,\mathsf{Skip} \longrightarrow \mathsf{Skip} \qquad \textsc{R-DEnd}\quad \mathsf{Dual}\,\mathsf{End} \longrightarrow \mathsf{End} \qquad \textsc{R-D?}\quad \mathsf{Dual}\,(?\,T) \longrightarrow !\,T \qquad \textsc{R-D!}\quad \mathsf{Dual}\,(!\,T) \longrightarrow ?\,T$$

$$\textsc{R-D\&}\quad \mathsf{Dual}\,(\&\{\overline{l_i : T_i}\}) \longrightarrow \oplus\{\overline{l_i : \mathsf{Dual}(T_i)}\} \qquad \textsc{R-D}\oplus\quad \mathsf{Dual}\,(\oplus\{\overline{l_i : T_i}\}) \longrightarrow \&\{\overline{l_i : \mathsf{Dual}(T_i)}\}$$

$$\textsc{R-DCtx}\quad \frac{T \longrightarrow U}{\mathsf{Dual}\,T \longrightarrow \mathsf{Dual}\,U} \qquad \textsc{R-DDVar}\quad \mathsf{Dual}\,(\mathsf{Dual}\,(\alpha\, T_1 \ldots T_m)) \longrightarrow \alpha\, T_1 \ldots T_m$$

Fig. 6: Type reduction.

Type formation $\boxed{\Delta \vdash T : \kappa}$

$$\textsc{K-Const}\quad \Delta \vdash \iota : \kappa_\iota \qquad \textsc{K-Var}\quad \frac{\alpha\colon \kappa \in \Delta}{\Delta \vdash \alpha : \kappa} \qquad \textsc{K-TAbs}\quad \frac{\Delta + \alpha\colon \kappa \vdash T : \kappa'}{\Delta \vdash \lambda\alpha\colon \kappa.T : \kappa \Rightarrow \kappa'} \qquad \textsc{K-TApp}\quad \frac{\Delta \vdash T : \kappa \Rightarrow \kappa' \quad \Delta \vdash U : \kappa \quad T\,U\ \mathsf{norm}}{\Delta \vdash T\,U : \kappa'}$$

Fig. 7: Type formation.

form types, which can be used in a compiler as well as in our proofs. We say that type T *normalises* to type U, written $T \Downarrow U$, if U whnf and U is reached from T in a finite number of reduction steps (note that any term which is already whnf normalises to itself). We write T norm to denote that $T \Downarrow U$ for some U.

For example, suppose we want to normalise the type $\mu_{\mathrm{s}}\, T$, where T is the type $\lambda v_1\colon \mathrm{s}.\oplus\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon !\alpha\}; \mathsf{Dual}\, v_1$. By computing all reductions from $\mu_{\mathrm{s}} T$, we obtain $\mu_{\mathrm{s}} T \longrightarrow T\,(\mu_{\mathrm{s}} T) \longrightarrow \oplus\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon !\alpha\}; \mathsf{Dual}\,(\mu_{\mathrm{s}} T) \not\longrightarrow$ for which we conclude that $\mu_{\mathrm{s}}\, T \Downarrow \oplus\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon !\alpha\}; \mathsf{Dual}\,(\mu_{\mathrm{s}} T)$. Similarly, we can reason that $\mu_{\mathrm{T}}\,(\lambda v_1\colon \mathrm{T}.v_1)$, $\mu_{\mathrm{s}}\,(\lambda v_1\colon \mathrm{s}.\mathsf{Skip}; v_1)$ and $\mu_{\mathrm{s}}\,(\lambda v_1\colon \mathrm{s}.\mathsf{Dual}\, v_1)$ are all examples of non-normalising expressions.

Equipped with normalisation, we can introduce *type formation*, which we do via the rules in Fig. 7. Rule K-Const introduces constants as types whose kinds match those of Fig. 4. Rule K-Var reads the kind of a type variable from context Δ. An abstraction $\lambda\alpha\colon \kappa.T$ is a well-formed type with kind $\kappa \Rightarrow \kappa'$ if T is well formed in context Δ updated with entry $\alpha\colon \kappa$ (rule K-TAbs). The update is necessary since we are dealing with renamed types and the same type variable may appear with different kinds in nested abstractions.

It is not until we reach rule K-TApp that we find a proviso about the normalisation of a type. This is standard and analogous to a condition on contractivity. The goal is to eliminate types that reduce indefinitely without reaching a whnf.

Theorem 1. *Let $\Delta \vdash T : \kappa$.*

Preservation. *If $T \longrightarrow U$, then $\Delta \vdash U : \kappa$.*
Confluence. *If $T \longrightarrow U$ and $T \longrightarrow V$, then $U \longrightarrow^* W$ and $V \longrightarrow^* W$.*
Weak normalisation. *$T \Downarrow U$ for some U. Furthermore, if $T \Downarrow V$, then $U = V$.*

We finally arrive at the main decidability result in this section. In its proof, we make use of the fact that recursion is restricted to kind $*$ to limit the possible subexpressions of the form $\mu_* U$ that might appear in the normalisation of T.

Theorem 2 (Decidability of type formation). *$\Delta \vdash T : \kappa$ is decidable for types in $F^{\mu_*;}_{\omega}$.*

4 Type equivalence

This section introduces type bisimulation as our notion of type equivalence. We define a labelled transition system (LTS) on the space of all types and write $T \xrightarrow{a} U$ to denote that T has a transition by label a to U. The grammar for labels and the LTS rules are in Fig. 8.

If T is not in weak head normal form, then we must normalise it to some type U, so that T has the same transitions as U (rule L-Red). Otherwise if $T\ \mathsf{whnf}$, then the transitions of T can be immediately derived by looking at the corresponding rule for T as follows. If T is a variable, use rule L-Var1 (with $m = 0$). If T is a constant (other than Skip), use rule L-Const. Note that if T is a lone Skip, then it has no transitions. If T is an abstraction, use rule L-Abs.

If T is an application, then we need to look inside the head. We write T as $T_0\ T_1 \ldots T_m$ with $m \geq 1$ where T_0 is not an application, and look at T_0. If T_0 is a variable, use rules L-Var1 and L-Var2. If T_0 is one of the constants $\rightarrow$, $\forall_\kappa$, $\odot\{\overline{l_i}\}$ or $(\!|\overline{l_i}|\!)$, use rule L-ConstApp. Note that T_0 is neither an abstraction nor μ_κ, since T is in weak head normal form. If T_0 is $\sharp$, we use rules L-Msg1 and L-Msg2. If T_0 is Dual, then the only way for T to be well-formed and in weak head normal form is if $m = 1$ and T_1 is α or $\alpha\ U_1 \ldots U_m$, in which case we use rules L-DualVar1 and L-DualVar2.

If T_0 is $;$, we require an additional case analysis on T_1. If $m = 1$, use rule L-Seq1. Otherwise $m = 2$ due to kinding. If T_1 is a variable, use rule L-VarSeq1 (with $m = 0$). If T_1 is a constant, then it must be of kind S. T_1 cannot be Skip, because T is in weak normal form, so it must be End, in which case we use rule L-EndSeq (End is an absorbing element, so $\mathsf{End}; U$ simply makes a transition to Skip without executing U). If T_1 is End. Note that T_1 cannot be an abstraction due to kinding.

$$a \;::=\; \alpha_i \;\mid\; \iota_i \;\mid\; \lambda\alpha\colon\kappa \qquad (i \geq 0, \iota \neq \mathsf{Skip}) \qquad \text{Transition labels}$$

Labelled transition system $\boxed{T \xrightarrow{a} U}$

$$\textsc{L-Red}\;\frac{T \longrightarrow U \quad U \xrightarrow{a} V}{T \xrightarrow{a} V} \qquad \textsc{L-Var1}\;\frac{m \geq 0}{\alpha\ T_1 \dots T_m \xrightarrow{\alpha_0} \mathsf{Skip}} \qquad \textsc{L-Var2}\;\frac{1 \leq j \leq m}{\alpha\ T_1 \dots T_m \xrightarrow{\alpha_j} T_j} \qquad \textsc{L-Const}\;\frac{\iota \neq \mathsf{Skip}}{\iota \xrightarrow{\iota_0} \mathsf{Skip}}$$

$$\textsc{L-ConstApp}\;\frac{\iota = \rightarrow, \forall_\kappa, \odot\{\overline{\ell_i}\}, (\!|\overline{\ell_i}|\!) \quad 1 \leq j \leq m}{\iota\ T_1 \dots T_m \xrightarrow{\iota_j} T_j} \qquad \textsc{L-Abs}\; \lambda\alpha\colon\kappa.T \xrightarrow{\lambda\alpha\colon\kappa} T \qquad \textsc{L-Msg1}\; \sharp T \xrightarrow{\sharp_1} T \qquad \textsc{L-Msg2}\; \sharp T \xrightarrow{\sharp_2} \mathsf{Skip}$$

$$\textsc{L-Seq1}\; ;\ T \xrightarrow{;_1} T \qquad \textsc{L-VarSeq1}\;\frac{m \geq 0}{(\alpha\ T_1 \dots T_m); U \xrightarrow{\alpha_0} U} \qquad \textsc{L-VarSeq2}\;\frac{1 \leq j \leq m}{(\alpha\ T_1 \dots T_m); U \xrightarrow{\alpha_j} T_j} \qquad \textsc{L-MsgSeq1}\; \sharp T; U \xrightarrow{\sharp_1} T$$

$$\textsc{L-MsgSeq2}\; \sharp T; U \xrightarrow{\sharp_2} U \qquad \textsc{L-ChoiceSeq}\;\frac{1 \leq j \leq m}{\odot\{\overline{\ell_i\colon T_i}\}; U \xrightarrow{\odot\{\overline{\ell_i}\}_j} T_j; U} \qquad \textsc{L-EndSeq}\; \mathsf{End}; U \xrightarrow{\mathsf{End}} \mathsf{Skip}$$

$$\textsc{L-DualVar1}\; \mathsf{Dual}\,(\alpha\ T_1 \dots T_m) \xrightarrow{\mathsf{Dual}_1} \alpha\ T_1 \dots T_m \qquad \textsc{L-DualVar2}\; \mathsf{Dual}\,(\alpha\ T_1 \dots T_m) \xrightarrow{\mathsf{Dual}_2} \mathsf{Skip}$$

$$\textsc{L-DualSeq1}\; (\mathsf{Dual}\,(\alpha\ T_1 \dots T_m)); U \xrightarrow{\mathsf{Dual}_1} \alpha\ T_1 \dots T_m \qquad \textsc{L-DualSeq2}\; (\mathsf{Dual}\,(\alpha\ T_1 \dots T_m)); U \xrightarrow{\mathsf{Dual}_2} U$$

Fig. 8: Labelled transition system for types.

If T_1 is an application, then again we write T_1 as $U_0\ U_1 \dots U_n$ with $n \geq 1$ where the head U_0 is not an application, and look at U_0. If U_0 is a variable, use rules L-VarSeq1 and L-VarSeq2. If U_0 is a constant, it must be one of $;$, μ_κ, $\sharp$, $\odot\{\overline{l_i}\}$ or Dual due to kinding. If U_0 is $\sharp$, use rules L-MsgSeq1 and L-MsgSeq2. If U_0 is $\odot\{\overline{l_i}\}$, use rule L-ChoiceSeq. If U_0 is Dual, the only way for T to be well-formed and in weak head normal form is if $n = 1$ and U_1 is α or $\alpha\ V_1 \dots V_\ell$, in which case we use rules L-DualSeq1 and L-DualSeq2. Note that U_0 cannot be $;$, μ_κ or an abstraction, since T is in weak normal form.

Let us clarify our LTS rules with an example. Consider the following type $\lambda\upsilon_1\colon \mathrm{T}.\mu\,\upsilon_2\colon \mathrm{S}.\oplus\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon !\upsilon_1\}; \mathsf{Dual}\,\upsilon_2$ and call it T. T is a type abstraction (on type variable υ_1), of kind $\mathrm{T} \Rightarrow \mathrm{S}$. It specifies a channel alternating between: offer a choice and output a value of type υ_1; or select a choice and input a value of type υ_1. The polarity is swapped thanks to the application of constant Dual to the recursion variable υ_2. To construct the (fragment of the) LTS generated by this type, let us first desugar T into $\lambda\upsilon_1\colon \mathrm{T}.U$ where U is the

$$\begin{tikzcd}
& \mathsf{Skip} & \lambda\upsilon_1 : \mathrm{T}.U \arrow[r, "\lambda\upsilon_1 : \mathrm{T}"] & U \arrow[dr, Rightarrow] & & \mathsf{End}; \mathsf{Dual}\, U \arrow[r, "\mathsf{End}"] & \mathsf{Skip} \\
& \upsilon_1 \arrow[u, "\upsilon_1"] & \mathsf{Dual}\,(\mathsf{Dual}\, U) \arrow[rr, Rightarrow] & & \oplus\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon !\upsilon_1\}; \mathsf{Dual}\, U \arrow[ur, "\oplus\{\mathsf{Done}, \mathsf{More}\}_1"'] \arrow[dr, "\oplus\{\mathsf{Done}, \mathsf{More}\}_2"] & & \\
& ?\upsilon_1; \mathsf{Dual}\,(\mathsf{Dual}\, U) \arrow[u, "?_1"] \arrow[ur, "?_2"'] & & & & !\upsilon_1; \mathsf{Dual}\, U \arrow[dl, "!_2"] \arrow[dr, "!_1"] & \\
& \&\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon ?\upsilon_1\}; \mathsf{Dual}\,(\mathsf{Dual}\, U) \arrow[u, "\&\{\mathsf{Done}, \mathsf{More}\}_2"] \arrow[d, "\&\{\mathsf{Done}, \mathsf{More}\}_1"'] & & & \mathsf{Dual}\, U \arrow[lll, Rightarrow] & & \upsilon_1 \arrow[d, "\upsilon_1"] \\
\mathsf{Skip} & \mathsf{End}; \mathsf{Dual}\,(\mathsf{Dual}\, U) \arrow[l, "\mathsf{End}"'] & & & & & \mathsf{Skip}
\end{tikzcd}$$

Fig. 9: The LTS for type $\lambda\upsilon_1 : \mathrm{T}.U$. Normalisation $T_1 \Downarrow T_2$ is represented as $T_1 \Rightarrow T_2$ and U is a shorthand for type $\mu_{\mathrm{S}}\,(\lambda\upsilon_2 : \mathrm{S}.\oplus\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon !\upsilon_1\}; \mathsf{Dual}\, \upsilon_2)$.

type $\mu_{\mathrm{S}}\,(\lambda\upsilon_2 : \mathrm{S}.\oplus\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon !\upsilon_1\}; \mathsf{Dual}\, \upsilon_2)$. Notice that U normalises to $\oplus\{\mathsf{Done}\colon \mathsf{End}, \mathsf{More}\colon !\upsilon_1\}; \mathsf{Dual}\, U$. The LTS for the example is sketched in Fig. 9. In this case, only finitely many types appear. However, more elaborate examples involving sequential composition or higher-order recursion may lead to an infinite graph of transitions.

Given the LTS rules, we can define, in the standard way, a notion of bisimulation. A binary relation R on types is called a *bisimulation* if, for every $(T, U) \in R$ and every transition label a:

1. if $T \xrightarrow{a} T'$, then there exists U' s.t. $U \xrightarrow{a} U'$ and $(T', U') \in R$;
2. if $U \xrightarrow{a} U'$, then there exists T' s.t. $T \xrightarrow{a} T'$ and $(T', U') \in R$.

We say that types T and U are bisimilar, written $T \sim U$, if there exists a bisimulation R such that $(T, U) \in R$.

Intuitively, a notion of type equivalence must preserve and reflect the syntax of type constructors: for example, a type $T \to U$ is equivalent to a type $T' \to U'$ iff T, T' are equivalent and U, U' are equivalent. Using the bisimulation technique, we achieve this by considering a labelled transition system on types: $T \to U$ has a transition labelled $\to_1$ to T and a transition labelled $\to_2$ to U. In this way, $T \to U$ can only be equivalent to another type which has two transitions with those same labels. For each of the type constructors ($\to$, $\forall_{\kappa}$, $!$, $?$, $\odot\{\overline{\ell_i}\}$, and so on) we have suitable transition rules. Moreover, a type sometimes needs to be reduced before a type constructor is found at the root of the syntax tree. If T normalizes to U, then we expect T and U to be bisimilar, which is achieved thanks to rule L-Red. This handles the various reductions: beta-reductions arising from lambda-abstraction and applications (e.g., $(\lambda\alpha\colon \kappa.T)\, U$ reduces to rename$(T[U/\alpha])$), reductions arising from the monoidal structure of sequential composition (e.g., $\mathsf{Skip}; T$ reduces to T), reductions arising from the internalisation of duality as a type constructor (e.g., $\mathsf{Dual}\,(!T)$ reduces to $?T$) and reductions arising from the recursion (e.g., $\mu_{\kappa}\, T$ reduces to $T\,(\mu_{\kappa}\, T)$).

Our notion of type equivalence enjoys natural properties and behaves as expected with respect to the notions of reduction, normalisation and kinding from Section 3. We can derive rules for type equivalence, that could be used to define another coinductive notion of equivalence, via effective syntax-directed rules. We can show that type equivalence is preserved under renaming, reduction and normalisation. We can also show that the axioms for sequential composition in the introduction (1) are derivable from our notion of bisimulation. These additional results are presented in the technical report [20].

5 Decidability of type equivalence

This section presents results on decidability of type equivalence. Our approach consists in translating types to objects in some computational model. We look at finite-state automata (for types in F^{μ}, $F^{\mu *}_{\omega}$, $F^{\mu \cdot}$, and $F^{\mu * \cdot}_{\omega}$), simple grammars (for types in $F^{\mu ;}$ and $F^{\mu * ;}_{\omega}$) and deterministic pushdown automata (for types in F^{μ}_{ω}, $F^{\mu \cdot}_{\omega}$ and $F^{\mu ;}_{\omega}$).

We say that a *grammar in Greibach normal form* is a tuple $(\mathcal{T}, \mathcal{N}, \gamma, \mathcal{R})$ where: $\mathcal{T}$ is a set of terminal symbols, denoted by a, b, c; $\mathcal{N}$ is a set of nonterminal symbols, denoted by X, Y, Z; $\gamma \in \mathcal{N}^*$ is the starting word; and $\mathcal{R} \subseteq \mathcal{N} \times \mathcal{T} \times \mathcal{N}^*$ is a set of productions. A grammar is said to be *simple* if, for every nonterminal X and every terminal a, there is at most one production $(X, a, \delta) \in \mathcal{R}$ [51].

Greek letters γ and δ denote (possibly empty) words of nonterminal symbols. Productions are written as $X \xrightarrow{a} \delta$. We define a notion of bisimulation for grammars via a labelled transition system. The system comprises a set of states $\mathcal{N}^*$ corresponding to words of nonterminal symbols. For each production $X \xrightarrow{a} \gamma$ and each word of nonterminal symbols δ, we have a labelled transition $X\delta \xrightarrow{a} \gamma\delta$. We let $\approx$ denote the bisimulation relation for grammars (the definition is similar to that in Section 4).

For the moment we focus on the class $F^{\mu * ;}_{\omega}$ and we explain how to convert a type T into a simple grammar $(\mathcal{T}_T, \mathcal{N}_T, \mathrm{word}(T), \mathcal{R}_T)$. The conversion is based on a function $\mathrm{word}(T)$ that maps each type T into a word of nonterminal symbols, while introducing fresh nonterminals and productions. In our construction, following the approach by Costa et al. [19], we use a nonterminal symbol with no productions, denoted by $\bot$, in order to separate the two descendants of a send/receive operation such as $!T;U$. The sequence of nonterminal symbols $\mathrm{word}(T)$ is defined as follows. First consider the cases in which T whnf.

- For any $m \geq 0$: $\mathrm{word}(\alpha\ T_1 \ldots T_m) = Y$ for Y a fresh nonterminal symbol with a production $Y \xrightarrow{\alpha_0} \varepsilon$ as well as $Y \xrightarrow{\alpha_j} \mathrm{word}(T_j)\bot$ for each $1 \leq j \leq m$.
- $\mathrm{word}(\mathsf{Skip}) = \varepsilon$.
- $\mathrm{word}(\mathsf{End}) = Y$ for Y a fresh symbol with a single production $Y \xrightarrow{\mathsf{End}} \bot$.
- for any $\iota \neq \mathsf{Skip}, \mathsf{End}$: $\mathrm{word}(\iota) = Y$ for Y a fresh nonterminal symbol with a single production $Y \xrightarrow{\iota} \varepsilon$.
- $\mathrm{word}(\lambda\alpha\colon \kappa.T) = Y$ for Y a fresh symbol with a production $Y \xrightarrow{\lambda\alpha\colon \kappa} \mathrm{word}(T)$.

- for any $m \geq 1$ and for ι one of $\rightarrow$, $\forall_{\kappa}$, $\odot\{\overline{l_i}\}$, $(\!|\overline{l_i}|\!)$: $\mathrm{word}(\iota\ T_1 \cdots T_m) = Y$ for a fresh nonterminal Y with a production $Y \xrightarrow{\iota_j} \mathrm{word}(T_j)$ for each $1 \leq j \leq m$.
- $\mathrm{word}(\sharp T) = Y$ for Y fresh with productions $Y \xrightarrow{\sharp_1} \mathrm{word}(T)\bot$ and $Y \xrightarrow{\sharp_2} \varepsilon$.
- $\mathrm{word}(;\ T) = Y$ for Y a fresh symbol with a production $Y \xrightarrow{;_1} \mathrm{word}(T)$.
- $\mathrm{word}(T;U) = \mathrm{word}(T)\,\mathrm{word}(U)$.
- $\mathrm{word}(\mathsf{Dual}\,(\alpha\ T_1 \ldots T_m)) = Y$ for Y a fresh symbol with productions $Y \xrightarrow{\mathsf{Dual}_1} \mathrm{word}(\alpha\ T_1 \ldots T_m)$ and $Y \xrightarrow{\mathsf{Dual}_2} \varepsilon$.

Finally, let us handle the cases where T is not in weak head normal form.

- If $T \Downarrow \mathsf{Skip}$, then $\mathrm{word}(T) = \varepsilon$.
- Otherwise if $T \Downarrow U \neq \mathsf{Skip}$, then $\mathrm{word}(T) = Y$ for Y a fresh nonterminal symbol. Let $Z\delta = \mathrm{word}(U)$. Then Y has a production $Y \xrightarrow{a} \gamma\delta$ for each production $Z \xrightarrow{a} \gamma$.

In the above construction, we create fresh symbols each time we encounter a weak head normal form other than Skip. In other words, $\mathcal{N}_T$ is the set containing $\bot$ and all nonterminals Y created during the computation of $\mathrm{word}(T)$. Another key insight is that the sequential composition of types is translated into a concatenation of words: $\mathrm{word}(T_1;T_2;\ldots;T_n) = \mathrm{word}(T_1)\,\mathrm{word}(T_2)\ldots\mathrm{word}(T_n)$. This allows our construction to terminate: even if the transitions lead to infinitely many types, they are split on the sequential composition operator, and so we only need to consider finitely many subexpressions.

For the last case in our construction to be well-defined, i.e., when $T \Downarrow U \neq \mathsf{Skip}$, we require $\mathrm{word}(U)$ to be non-empty. Indeed, if $U\,\mathsf{whnf}$, then we can observe (by inspecting all cases) that $\mathrm{word}(U) = \varepsilon$ iff $U = \mathsf{Skip}$. We also need to argue that the construction of $\mathrm{word}(T)$ eventually terminates. For this, we keep track of all types visited during the construction, and we only add a fresh nonterminal Y to our grammar if the type visited is syntactically different from all types visited so far. Therefore, we reuse the same symbol Y with the same productions each time we revisit a type. With all these observations, we get the following result.

Lemma 1. *Suppose that $T \in F^{\mu *;}_{\omega}$. Then the construction of* $\mathrm{word}(T)$ *terminates producing a simple grammar.*

We illustrate the above construction with the polymorphic tree exchanging example from Section 2,

```
type TreeC a = &{Leaf: Skip, Node: TreeC a; ?a ; TreeC a}
```

that is written in $F^{\mu *;}_{\omega}$ as $T_0 = \lambda v_1 : \mathrm{T}.\mu\, v_2 : \mathrm{S}.\,\&\{\mathsf{Leaf}: \mathsf{Skip}, \mathsf{Node}: v_2; ?v_1; v_2\}$. For ease of notation, in this example we write $\&_i$ as shorthand for $\&\{\mathsf{Leaf}, \mathsf{Node}\}_i$. Since T_0 is in weak head normal form, $\mathrm{word}(T_0)$ returns a fresh symbol, which we call X_0. We also have a production $X_0 \xrightarrow{\lambda v_1 : \mathrm{T}} \mathrm{word}(T_1)$, where T_1 is the type $\mu\, v_2 : \mathrm{S}.\,\&\{\mathsf{Leaf}: \mathsf{Skip}, \mathsf{Node}: v_2; ?v_1; v_2\}$. Since T_1 is not in whnf, we must normalise it, to get $T_2 = \&\{\mathsf{Leaf}: \mathsf{Skip}, \mathsf{Node}: T_1; ?v_1; T_1\}$. Therefore $\mathrm{word}(T_1)$ returns a fresh symbol, which we call X_1. To obtain the transitions of X_1, we must first compute $\mathrm{word}(T_2)$, which is a fresh symbol X_2 with transitions $X_2 \xrightarrow{\&_1} \mathrm{word}(\mathsf{Skip})$ and $X_2 \xrightarrow{\&_2} \mathrm{word}(T_1; ?\upsilon_1; T_1)$. Thus we also get $X_1 \xrightarrow{\&_1} \mathrm{word}(\mathsf{Skip})$ and $X_1 \xrightarrow{\&_2} \mathrm{word}(T_1; ?\upsilon_1; T_1)$.

We have $\mathrm{word}(\mathsf{Skip}) = \varepsilon$, but we still need to compute $\mathrm{word}(T_1; ?\upsilon_1; T_1)$. This type normalises to $T_3 = T_2; ?\upsilon_1; T_1$ since $T_1 \Downarrow T_2$. Thus $\mathrm{word}(T_1; ?\upsilon_1; T_1)$ is a fresh symbol X_3. To obtain the productions of X_3 we must compute $\mathrm{word}(T_2; ?\upsilon_1; T_1) = \mathrm{word}(T_2)\,\mathrm{word}(?\upsilon_1)\,\mathrm{word}(T_1)$. At this point we already have $\mathrm{word}(T_1) = X_1$ and $\mathrm{word}(T_2) = X_2$. We still need to compute $\mathrm{word}(?\upsilon_1)$, which is a fresh symbol X_4 with productions $X_4 \xrightarrow{?_1} \mathrm{word}(\upsilon_1)\bot$ and $X_4 \xrightarrow{?_2} \varepsilon$. In turn, $\mathrm{word}(\upsilon_1)$ is a fresh symbol X_5 with a production $X_5 \xrightarrow{\upsilon_1} \varepsilon$. Finally, we get $\mathrm{word}(T_2; ?\upsilon_1; T_1) = X_2X_4X_1$, which means we can write the productions for X_3: $X_3 \xrightarrow{\&_1} X_4X_1$ and $X_3 \xrightarrow{\&_2} X_3X_4X_1$.

Putting all this together, we can finally obtain the simple grammar:

$$\begin{array}{ccccc}
X_0 \xrightarrow{\lambda\upsilon_1:\mathsf{T}} X_1 & X_1 \xrightarrow{\&_1} \varepsilon & X_1 \xrightarrow{\&_2} X_3 & X_2 \xrightarrow{\&_1} \varepsilon & X_2 \xrightarrow{\&_2} X_3 \\
X_3 \xrightarrow{\&_1} X_4X_1 & X_3 \xrightarrow{\&_2} X_3X_4X_1 & X_4 \xrightarrow{?_1} X_5\bot & X_4 \xrightarrow{?_2} \varepsilon & X_5 \xrightarrow{\upsilon_1} \varepsilon
\end{array}$$

Next, we argue that type equivalence (i.e., bisimilarity on types) corresponds to bisimilarity on the corresponding grammars. This is achieved by the following lemma, that asserts that the LTS of a type and the LTS of the corresponding word of nonterminals have exactly the same transitions.

Lemma 2 (Full abstraction). *Let $T \in F^{\mu*;}_{\omega}$ and $(\mathcal{T}_T, \mathcal{N}_T, \mathrm{word}(T), \mathcal{R}_T)$ the corresponding simple grammar. Suppose also that $\mathrm{word}(T) \approx \gamma$.*

1. *If $T \xrightarrow{a} U$ then there exists γ' such that $\gamma \xrightarrow{a} \gamma'$ and $\mathrm{word}(U) \approx \gamma'$.*
2. *If $\gamma \xrightarrow{a} \gamma'$ then there exists U such that $T \xrightarrow{a} U$ and $\mathrm{word}(U) \approx \gamma'$.*

As a consequence of the above result, we get soundness and completeness of the bisimilarity $\mathrm{word}(T) \approx \mathrm{word}(U)$ with respect to the bisimilarity $T \sim U$. Indeed by Lemma 2, any sequence of transitions starting from T can be matched by a sequence of transitions starting from $\mathrm{word}(T)$; and similarly for U. Thus $T \sim U$ iff $\mathrm{word}(T) \approx \mathrm{word}(U)$.

Theorem 3. *The type equivalence problem is decidable for types in $F^{\mu*;}_{\omega}$.*

For the remainder of this section, we look at the other classes of types in Fig. 2 and examine the computation models they correspond to. Since class $F^{\mu;}$ is contained in $F^{\mu*;}_{\omega}$, we can express types without λ-abstractions with simple grammars as well. In this way we recover previous results in the literature [4,19].

Let us now look at the class $F^{\mu*\cdot}_{\omega}$. In this class we do not have Skip nor sequential composition and message operators are binary ($\sharp T.U$) rather than unary. Since we do not have sequential composition, there is no need to consider words of nonterminals, and instead it suffices to translate types into single symbols, i.e., states in an automaton. Moreover, since there is no recursion beyond μ_κ, only finitely many types can be reached from a given T. We can thus adapt our construction as follows for $F^{\mu*\cdot}_{\omega}$. In the definition of the LTS (Fig. 8):

- discard all rules involving sequential composition;
- discard rules L-Var1 for $m > 0$ and L-DualVar2 (they were only needed to distinguish types in sequential composition);
- discard case $\iota = \mathsf{End}$ in rule L-Const (so that End no longer has transitions);
- replace Skip with End on the right-hand side of rules L-Var1 with $m = 0$ and L-Const;
- discard rules L-Msg1 and L-Msg2 and treat $\iota = \sharp$ like the other constants in rule L-ConstApp.

Also replace the construction of word(T) into a construction of state(T), associating to each type T a state in a finite-state automata. For each transition $T \xrightarrow{a} U$ we have the corresponding transition state(T) $\xrightarrow{a}$ state(U). Notice that the resulting automata is deterministic since the original LTS is also deterministic (for each type T and label a, there is at most one transition $T \xrightarrow{a} U$). Since bisimilarity of deterministic finite-state automata can be decided in polynomial time [44], we get the following results.

Theorem 4.

1. *To each type T in $F^{\mu_*\cdot}_{\omega}$ we can associate a finite-state automata corresponding to the (fragment of the) LTS generated by T.*
2. *The type equivalence problem is polynomial-time decidable for types in $F^{\mu_*\cdot}_{\omega}$.*

Clearly, Theorem 4 applies to the subclasses of $F^{\mu_*\cdot}_{\omega}$: F^{μ}, $F^{\mu\cdot}$ and $F^{\mu_*}_{\omega}$. In this way we recover previous results in the literature [14,19,33].

Finally, we consider the classes F^{μ}_{ω}, $F^{\mu\cdot}_{\omega}$ and $F^{\mu;}_{\omega}$ involving arbitrarily-kinded recursion. We shall show that these classes are already powerful enough to simulate deterministic pushdown automata; hence, the type equivalence problem becomes impractical (i.e., no practical implementation of an algorithm is known). We only focus on the simplest case F^{μ}_{ω}, as the others two classes are even more expressive. Instead of looking at deterministic pushdown automata, we look at deterministic first-order grammars, which constitute an equivalent model of computation [46]. This choice simplifies our construction. We say that a *first-order grammar* is a tuple $(\mathcal{X}, \mathcal{T}, \mathcal{N}, E, \mathcal{R})$ where:

- $\mathcal{X}$ is a set of variables $\alpha, \beta, \ldots$; $\mathcal{T}$ is a set of terminal symbols $a, b, \ldots$; $\mathcal{N}$ is a set of nonterminal symbols $X, Y, \ldots$
- each nonterminal X has an arity $m = \text{arity}(X) \in \mathbb{N}$.
- the set $\mathcal{E}$ of expressions over $\mathcal{X}$, $\mathcal{N}$ is inductively defined by two rules: any variable α is an expression; if arity(X) $= m$ and $E_1, \ldots, E_m$ are expressions, then so is $X\ E_1 \ldots E_m$. Whenever $m = 0$, X is called a constant.
- E is an expression over $\mathcal{N}$, called the initial expression.
- $\mathcal{R}$ is a set of productions. Each production is a triple (X, a, E), written as $X\ \alpha_1 \ldots \alpha_m \xrightarrow{a} E$, where $m = \text{arity}(X)$ and the variables in E must be taken from $\alpha_1, \ldots, \alpha_m$.

A first-order grammar is *deterministic* if, for every X and a, there is at most one production $(X, a, E) \in \mathcal{R}$.

Just as a simple grammar defines an LTS over words of nonterminals, a first-order grammar defines an LTS over the set $\mathcal{E}_0$ of closed expressions. For each production $X\ \alpha_1 \dots \alpha_m \xrightarrow{a} E$ we have the labelled transition $X\ E_1 \dots E_m \xrightarrow{a} E[E_1/\alpha_1, \dots, E_m/\alpha_m]$.

Let $\approx$ denote bisimilarity over closed expressions according to a first-order grammar. We now present a fully abstract (i.e., preserving bisimilarity) translation of a deterministic first-order grammar into a type in F^{μ}_{ω}. Each grammar variable α has a corresponding type variable α (of kind T). An expression $X\ E_1 \dots E_m$ is represented as a type application $X\ E_1 \dots E_m$. If X has arity m and the productions $X\ \alpha_1 \dots \alpha_m \xrightarrow{a_j} E_j$ for a range of j, then we write the equation specifying X as a record (since the first-order grammar is deterministic, all record labels are distinct, and thus the right-hand side on the equation specifying X is well-formed).

$$X \doteq \lambda\alpha_1 \colon \mathrm{T}. \dots \lambda\alpha_m \colon \mathrm{T}.\{a_1 \colon E_1, \dots, a_m \colon E_m\}$$

This gives rise to a system of equations $\{X_i \doteq T_i\}$, one for each nonterminal X_i, where the nonterminals may appear in the right-hand sides T_i. Finally, given an initial expression E, it is standard how to convert it into a μ-type using the system above.

Using the above translation, we are able to simulate a transition $E \xrightarrow{a_j} F$ of the first-order grammar as a transition $E \xrightarrow{\{\overline{a_i}\}_j} F$ on the corresponding types. Therefore, the translation is fully abstract and we get the following result.

Theorem 5. *Let E and F be closed expressions on a first-order grammar and E, F the corresponding types. Then $E \approx F$ iff $E \sim F$.*

Let us work on an example to better understand the above translation. Consider the language $L_3 = \{\ell^n a r^n a \mid n \geq 0\} \cup \{\ell^n b r^n b \mid n \geq 0\}$ over the alphabet $\{a, b, \ell, r\}$. L_3 is a typical example of a language that cannot be described with a simple grammar, but can be accepted by a deterministic pushdown automaton [51]. Consider the first-order grammar with nonterminals $X, R, A, B, \perp$, initial expression $X\ A\ B$, and productions

$$X\ \alpha\ \beta \xrightarrow{\ell} X\ (R\ \alpha)\ (R\ \beta) \qquad X\ \alpha\ \beta \xrightarrow{a} \alpha \qquad X\ \alpha\ \beta \xrightarrow{b} \beta$$
$$R\ \alpha \xrightarrow{r} \alpha \qquad A \xrightarrow{a} \perp \qquad B \xrightarrow{b} \perp$$

Note that $\perp$ is a constant without productions. It is easy to see that the traces of this first-order grammar correspond exactly to the words in L_3. By following the steps in the above translation, we arrive at the system of equations

$$X \doteq \lambda\alpha \colon \mathrm{T}.\lambda\beta \colon \mathrm{T}.\{\ell \colon X(R\alpha)(R\beta), a \colon \alpha, b \colon \beta\} \qquad R \doteq \lambda\alpha \colon \mathrm{T}.\{r \colon \alpha\}$$
$$A \doteq \{a \colon \perp\} \qquad B \doteq \{b \colon \perp\} \qquad \perp \doteq \{\}$$

Therefore, the initial expression $X\ A\ B$ becomes the type

$$(\mu\,\xi \colon \mathrm{T} \Rightarrow \mathrm{T} \Rightarrow \mathrm{T}.\ \lambda\alpha \colon \mathrm{T}.\lambda\beta \colon \mathrm{T}.\{\ell \colon \xi\{r \colon \alpha\}\{r \colon \beta\}, a \colon \alpha, b \colon \beta\})\{a \colon \{\}\}\{b \colon \{\}\},$$

whose transitions simulate the transitions of the first-order grammar.

$$
\begin{array}{rcl}
v & ::= & c \mid x \mid \lambda x\colon T.t \mid \mathsf{rec}\, x\colon T.v \mid \Lambda\alpha\colon \kappa.v \mid \{\overline{l_i = v_i}\} \mid \langle l = v\rangle\, \mathsf{as}\, T \\
 & & \mathsf{receive}[T] \mid \mathsf{receive}[T][T] \mid \mathsf{send}[T] \mid \mathsf{send}[T]\, v \mid \mathsf{send}[T]\, v[T] \\
t & ::= & v \mid t\,t \mid t[T] \mid \{\overline{l_i = t_i}\} \mid \mathsf{let}\, \{\overline{l_i = x_i}\} = t\, \mathsf{in}\, t \\
 & & \langle l = t\rangle\, \mathsf{as}\, T \mid \mathsf{case}\, t\, \mathsf{of}\, t \mid \mathsf{match}\, t\, \mathsf{with}\, t \\
p & ::= & \langle t\rangle \mid p \mid p \mid (\nu x x)p
\end{array}
$$

c ::=		Term constant
$\mathsf{receive}$	$\forall\alpha\colon \mathrm{T}.\, \forall\beta\colon \mathrm{S}.\, ?\alpha.\beta \to \alpha \otimes \beta$	receive on a channel
send	$\forall\alpha\colon \mathrm{T}.\, \alpha \to \forall\beta\colon \mathrm{S}.\, !\alpha.\beta \to \beta$	send on a channel
$\mathsf{select}\; l_j\; \mathsf{as} \oplus\{\overline{l_i\colon T_i}\}$	$\oplus\{\overline{l_i\colon T_i}\} \to T_j$	internal choice
close	$\mathsf{End} \to \mathsf{Unit}$	channel close
fork	$(\mathsf{Unit} \to \mathsf{Unit}) \to \mathsf{Unit}$	fork a new thread
new	$\forall\alpha\colon \mathrm{S}.\, a \to \alpha \otimes \mathsf{Dual}\,\alpha$	channel creation

Fig. 10: Terms and types for term constants.

6 The term language and its metatheory

This section briefly introduces a concurrent functional language equipped with $F^{\mu_*;}_{\omega}$ types, together with its metatheory. The results mostly follow from those in the literature, although explicit recursion at the term level and the unrestricted bindings in typing contexts are somewhat new in session types. The complete set of rules is to be found in the technical report [20].

The syntax of terms and processes is defined by the grammar in Fig. 10. The same figure introduces types for the constants. The term language is essentially the polymorphic lambda calculus with support for session operators, formulated as in Almeida et al. and Cai et al. [2,14]. From System F it comprises terms and type abstractions, records and variants, including constructors and destructors in each case. The support for session operations and concurrency includes channel creation (new), the different channel operations (receive, send, match, select and close) and thread creation (fork). We program at the term level and use processes only for the runtime. Processes include terms as threads, parallel composition and channel creation, all inspired in the pi-calculus with double binders [73].

Process typing and an excerpt of term typing is in Fig. 11. A judgement of the form $\Delta \mid \Gamma \vdash t\colon T$ records the fact that term t has type T under contexts Δ (recording kinds for type variables) and Γ (recording types for term variables). The judgement for processes, $\Gamma \vdash p$, says that p is well-typed under context Γ. It simplifies that for terms, since processes feature no free type variables and are assigned no particular type. Once again, the rules are adapted from the two above cited works. The difference to Cai et al. is that we work in a linear setting and hence axioms (T-Const and T-Var) work on an empty context, and most of the other rules must split the context accordingly. Rule T-TAbs simplifies

Term typing $\boxed{\Delta \mid \Gamma \vdash t\colon T}$

$$\textsc{T-Const}\ \frac{\Delta \vdash T_c : *}{\Delta \mid \cdot \vdash c\colon T_c} \qquad \textsc{T-Var}\ \Delta \mid x\colon T \vdash x\colon T \qquad \textsc{T-App}\ \frac{\Delta \mid \Gamma_1 \vdash t_1\colon U \to T \qquad \Delta \mid \Gamma_2 \vdash t_2\colon U}{\Delta \mid \Gamma_1, \Gamma_2 \vdash t_1\, t_2\colon T}$$

$$\textsc{T-Rec}\ \frac{\Delta \vdash T : * \qquad \Delta \mid \Gamma, x :^{\omega} T \to U \vdash v\colon T \to U}{\Delta \mid \Gamma \vdash \mathsf{rec}\, x\colon T \to U.v\colon T \to U} \qquad \textsc{T-TAbs}\ \frac{\Delta, \alpha\colon \kappa \mid \Gamma \vdash v\colon T\,\alpha}{\Delta \mid \Gamma \vdash (\Lambda\alpha\colon \kappa.v)\colon \forall_{\kappa}\, T}$$

$$\textsc{T-Match}\ \frac{\Delta \mid \Gamma_1 \vdash t_1 : \&\{\overline{l_i : T_i}\} \qquad \Delta \mid \Gamma_2 \vdash t_2 : \{\overline{l_i\colon T_i \to T}\}}{\Delta \mid \Gamma_1, \Gamma_2 \vdash \mathsf{match}\, t_1\, \mathsf{with}\, t_2 : T} \qquad \textsc{T-Eq}\ \frac{\Delta \mid \Gamma \vdash t\colon U \qquad \Delta \vdash U : * \qquad U \sim T}{\Delta \mid \Gamma \vdash t\colon T}$$

$$\textsc{T-Dereliction}\ \frac{\Delta \mid \Gamma, x\colon T \vdash t\colon U}{\Delta \mid \Gamma, x :^{\omega} T \vdash t\colon U} \qquad \textsc{T-Weakening}\ \frac{\Delta \mid \Gamma \vdash t\colon U}{\Delta \mid \Gamma, x :^{\omega} T \vdash t\colon U} \qquad \textsc{T-Contraction}\ \frac{\Delta \mid \Gamma, y :^{\omega} T, z :^{\omega} T \vdash t\colon U}{\Delta \mid \Gamma, x :^{\omega} T \vdash t[x/y][x/z]\colon U}$$

Process typing $\boxed{\Gamma \vdash p}$

$$\frac{\varepsilon \mid \Gamma \vdash t\colon \mathsf{Unit}}{\Gamma \vdash \langle t \rangle} \qquad \frac{\Gamma_1 \vdash p_1 \qquad \Gamma_2 \vdash p_2}{\Gamma_1, \Gamma_2 \vdash p_1 \mid p_2} \qquad \frac{\Gamma, x\colon T, y\colon \mathsf{Dual}\, T \vdash p}{\Gamma \vdash (\nu x y) p}$$

Fig. 11: Typing (excerpt).

that of Cai et al.; we can easily show that both rules are interchangeable. We support exponentials [37] for recursive functions, so that one may write functions that feature more than one recursive call (good for consuming binary trees, for example) and branches that do not use the recursive function (for code that is supposed to terminate). Towards this end, we add an unrestricted binding $x :^{\omega} T$ in term variable contexts, an explicit rule for rec (as opposed to making rec a constant as in Cai et al. [14]) and substructural rules for unrestricted bindings (T-Dereliction, T-Weakening and T-Contraction).

Thanks to the power of System F, most of the session and concurrency operators are expressed as constants. For example, $\mathsf{receive}$ receives a session type $!\alpha.\beta$ with α, the payload of the message, an arbitrary type and β, the continuation, a session type, and returns a pair of the value received and the continuation channel. As usual $\forall\alpha\colon \kappa.T$ abbreviates the type $\forall_{\kappa}\, (\lambda\alpha\colon \kappa.T)$. The exception is the external choice (T-Match) which can not be captured by a type (similarly to T-Case) and hence requires a dedicated typing rule.

Process reduction is in Fig. 12. Following Milner [55] we factor out processes by means of a structural congruence relation that accounts for the associative and commutative nature of parallel composition, scope extrusion and exchanging the order of channel bindings.We now address the metatheory of our language, starting with preservation for both terms and processes.

Process reduction $\boxed{p \to p}$

$$\frac{t_1 \to t_2}{\langle t_1 \rangle \to \langle t_2 \rangle} \qquad \langle E[\mathsf{fork}\, v] \rangle \to \langle E[\{\}] \rangle \mid \langle v\,\{\} \rangle \qquad \langle E[\mathsf{new}[T]] \rangle \to (\nu xy)\langle E[\{x, y\}] \rangle$$

$$(\nu xy)(\langle E_1[\mathsf{receive}[T][U]\, y] \rangle \mid \langle E_2[\mathsf{send}[V][W]\, v\, x] \rangle) \to (\nu xy)(\langle E_1[\{y, v\}] \rangle \mid \langle E_2[x] \rangle)$$

$$(\nu xy)(\langle E_1[\mathsf{match}\, y\, \mathsf{with}\, \{\overline{l_i = t_i}\}] \rangle \mid \langle E_2[(\mathsf{select}\, l_j\, \mathsf{as}\, T)\, x] \rangle) \to (\nu xy)\langle E_1[t_j\, y] \rangle \mid \langle E_2[x] \rangle$$

$$(\nu xy)(\langle E_1[\mathsf{close}\, y] \rangle \mid \langle E_2[\mathsf{close}\, x] \rangle) \to \langle E_1[\{\}] \rangle \mid \langle E_2[\{\}] \rangle \qquad \frac{p_1 \to p_2}{p_1 \mid q \to p_2 \mid q}$$

$$\frac{p_1 \to p_2}{(\nu xy)p_1 \to (\nu xy)p_2} \qquad \frac{p_1 \equiv p_2 \qquad p_2 \to p_3 \qquad p_3 \equiv p_4}{p_1 \to p_4}$$

Fig. 12: Process reduction.

Theorem 6 (Preservation).

1. *If $\Delta \mid \Gamma \vdash t : T$ and $t \to t'$, then $\Delta \mid \Gamma \vdash t' : T$.*
2. *If $\Gamma \vdash p$ and $p \equiv p'$, then $\Gamma \vdash p'$.*
3. *If $\Gamma \vdash p$ and $p \to p'$, then $\Gamma \vdash p'$.*

Progress for the term language is assured only when the typing context contains channel endpoints only. When Δ is understood from the context we write Γ^{s} to mean that Γ contains only types of kind s, that is $\Delta \vdash T : \mathsf{s}$ for all types T in Γ. Well typed terms are values, or else they may reduce or are ready to reduce at the process level. Reduction in the case of session operations—$\mathsf{receive}$, send, match, select, close—is pending a matching counterpart.

Theorem 7 (Progress for the term language). *If $\Delta \mid \Gamma^{\mathsf{s}} \vdash t : T$, then t is a value, t reduces, or t is stuck in one of the following forms: $E[\mathsf{fork}\, v]$, $E[\mathsf{new}[T]]$, $E[\mathsf{receive}[T][U]\, v]$, $E[\mathsf{send}[U]\, T[v]\, x]$, $E[\mathsf{match}\, y\, \mathsf{with}\, \{\overline{l_i = t_i}\}]$, $E[(\mathsf{select}\, l_j\, \mathsf{as}\, T)\, x]$, or $E[\mathsf{close}\, x]$.*

In order to state our result on the absence of runtime errors we need a few notions on the structure of terms and processes; here we follow Almeida et al. [2]. The *subject* of an expression e, denoted by subj(e), is x in the following cases.

$$\mathsf{receive}[T][U]\, x \quad \mathsf{send}[T]\, v[U]\, x \quad \mathsf{match}\, x\, \mathsf{with}\, t \qquad (\mathsf{select}\, l_j\, \mathsf{as}\, T)\, x \qquad \mathsf{close}\, x$$

Two terms e_1 and e_2 *agree* on channel xy, notation $\mathrm{agree}^{xy}(e_1, e_2)$, in the following cases (symmetric forms omitted).

$$\mathrm{agree}^{xy}(\mathsf{receive}[T][U]\, x, \mathsf{send}[V]\, v[W]\, y) \qquad \mathrm{agree}^{xy}(\mathsf{close}\, x, \mathsf{close}\, y)$$
$$\mathrm{agree}^{xy}(\mathsf{match}\, x\, \mathsf{with}\, \{\overline{l_i = t_i}\}_{i \in I}, (\mathsf{select}\, l_j\, \mathsf{as}\, T)\, y) \quad j \in I$$

A closed process is a *runtime error* if it is structurally congruent to some process that contains a subexpression or subprocess of one of the following forms.

1. $v\,u$ where v is not a λ or a rec and $v \neq \mathsf{receive}[T][U]$, $\mathsf{send}[T]\,u$, $\mathsf{send}[T]\,w[U]$, $\mathsf{select}\,l_j\,\mathsf{as}\,T$, close, fork;
2. $v[T]$ where v in not a Λ and $v \neq \mathsf{receive}$, $\mathsf{receive}[U]$, send, $\mathsf{send}[U]$, new;
3. $\mathsf{let}\,\{\overline{l_i = x_i}\} = v\,\mathsf{in}\,t$ and v is not of the form $\{\overline{l_i = u_i}\}$;
4. $\mathsf{case}\,v\,\mathsf{of}\,t$ and $v \neq \langle l_j = u\rangle\,\mathsf{as}\,T$ or $t \neq \{\overline{l_i = u_i}\}_{i\in I}$ with $j \notin I$;
5. $\mathsf{receive}[T][U]\,v$ or $\mathsf{send}[T]\,u[U]\,v$ or $\mathsf{match}\,v\,\mathsf{with}\,t$ or $(\mathsf{select}\,l\,\mathsf{as}\,T)\,v$ or $\mathsf{close}\,v$ and v is not an endpoint x;
6. $\langle E_1[e_1]\rangle \mid \langle E_2[e_2]\rangle$ and $\mathrm{subj}(e_1) = \mathrm{subj}(e_2)$;
7. $(\nu xy)(\langle E_1[e_1]\rangle \mid \langle E_2[e_2]\rangle)$ and $\mathrm{subj}(e_1) = x$, $\mathrm{subj}(e_2) = y$, $\neg\,\mathrm{agree}^{xy}(e_1, e_2)$.

The four cases are standard to system F with records and variants. The support for session types and concurrency in the first two cases (term and type application) are derived from the types of values for such operators (Fig. 10). Item 5 addresses session operators applied to non endpoints. Item 6 is for two concurrent session operators on the same channel end. Finally, Item 7 is for mismatches on two session operations on two endpoints for the same channel.

Theorem 8 (Safety). *If $\Gamma^{\mathsf{s}} \vdash p$, then p is not a runtime error.*

An *algorithmic typing system* can be easily extracted from the declarative system for terms in Fig. 11 via a bidirectional type system, formulated along the lines of Almeida et al. [2].

7 Related Work

Equirecursion in system F. In first investigations on equirecursive types, the notion of type equivalence is often formulated in a coinductive fashion [5,11,18,29,38]. Two types are equivalent if they unroll into the same infinite tree. Whenever this unrolling is the only type-level computation, such trees are regular, enabling efficient decision procedures. Some authors have studied equirecursion together with other notions of type-level computation. Solomon considers parameterized type definitions, which correspond to higher-order kinds [63]. These implicitly correspond to λ-terms, since reduction occurs as types are allowed to call other types. Some authors consider equirecursion in system F_ω, with weaker or stronger notions of equality [1,12,14,41]. Regarding equirecursion in system F, the model of Cai et al. [14] is the closest to ours, and indeed our results up to $F_\omega^{\mu_*\cdot}$ can be seen as a generalisation of theirs. However, Cai et al. depart from the usual setting by allowing non-contractive types (which most authors forbid, including this work), requiring a sort of infinitary lambda calculus. Moreover, this work further extends additional equivalence properties by including session types with their distinctive semantics, such as sequential composition and duality.

Session type systems. Session types were introduced in the 90s by Honda et al. [42,43,67]. Equirecursion was the first approach used to construct infinite session types, which often allows type equality to be interpreted according to a coinductive notion of bisimulation [52]. In this vein, Keizer et al. [48] utilize coalgebras to represent session types. Since the inception of session types, there has been an interest in extending the theory to nonregular protocols [58,59,66]. Context-free session types emerged as a natural extension, as it still allowed for practical type equality algorithms [3,4,19,28,56,68]. Other approaches that go beyond regular session types include nested session types [24] as well as 1-counter, pushdown and 2-counter session types [33]. However, the more expressive notions are not amenable to practical type equivalence algorithms, just like the higher-order types present in our system F_ω^μ. Polymorphism in session types has also been a topic of interest, with or without recursion [15,22,23,31,39].

Dual type operator. This work is, to the best of our knowledge, the first that internalises duality as a type constructor. Other settings, such as the language Alms [72], consider duality for session types as a user-definable, not built in, type function. Our Dual is a type operator, not a type function. The difference is that a type function involves a type-level computation, which converges to a type written without dual. For example, in Alms we would have $\mathsf{dual}(!\mathsf{Int.End}) = ?\mathsf{Int.End}$ (as a type-level computation), both sides being the *same* type. In our setting, $\mathsf{Dual}\,(!\mathsf{Int}; \mathsf{End})$ is a type on its own, which happens to be equivalent to $?\mathsf{Int}; \mathsf{End}$. At the same time, our setting allows for types such as $\mathsf{Dual}\,\alpha$, or $(\mathsf{Dual}\,\alpha); T_1; T_2$, which do not reduce.

Type equivalence algorithms. Algorithms for deciding the equivalence of types must inherently be related to the computational power of the corresponding type system. This has been used implicitly or explicitly to obtain decidability results. As already explained, if equirecursion is the only type-level computation, types can be represented as finite-state automata (or equivalently, infinite regular trees). Although some exponential time algorithms were first proposed [32], it has been established that the problem can be solved in quadratic time [53], which is to be expected as it matches the corresponding problem of bisimulation of finite-state automata [44]; see also Pierce [57].

The next ‘simplest’ model of computation is that of simple grammars, which intuitively correspond to deterministic pushdown automata with a single state [33]. Almeida et al. [4] provided a practical algorithm for checking the bisimilarity of simple grammars. By dropping the determinism assumption, we arrive at Greibach normal form grammars, which are equivalent to basic process algebras [6,7]. Bisimilarity algorithms have been studied extensively in this setting [13,17,47,49]; presently it is known that the complexity of the problem lies between EXPTIME and 2-EXPTIME, which does not exclude the possibility of a polynomial time algorithm for the simpler model of simple grammars.

In this paper we present a reduction from first-order grammars to F_ω^μ-types, showing that the more expressive type systems (F_ω^μ, presented here and in Cai et al. [14], as well as its extensions) are at least as powerful as deterministic pushdown automata. As far as we know, the closest result to ours is by Solomon [63], which shows conversions between a universe of “context-free types” and deterministic context-free languages. The universe of types studied by Solomon is different from F_ω^μ. With some work we could prove that Solomon’s types can be embedded into F_ω^μ, which would entail our result as a corollary. However, it is easier and simpler to prove directly the reduction as we did.

The equivalence problem for deterministic pushdown automata was a notorious open problem for a long time, until Sénizergues showed it to be decidable [61,62]. Since his proof, many authors have tried to refine the result in an attempt to arrive at an implementable algorithm [46,64,65].

Concurrent term languages. The usefulness of a type system is directly related to its capability to be used in a programming language. Type systems such as the ones discussed in this work lend themselves quite readily to functional term languages [45]. For session types, existing term languages are either inspired in the pi calculus [26,73,69] or in the lambda calculus [35,54,70], or even the two [71]. The system presented in this paper is linear, meaning that resources must be used exactly once [50,74]. Some authors go beyond linearity by considering unrestricted type qualifiers [48,73] or manifest sharing [8].

8 Conclusion and future work

This paper introduces an extension of system F which includes equirecursion, lambda abstractions, and context-free session types. We present type equivalence algorithms, and a term language and its metatheory. Although we have defined a rather general system, it turns out that for practical purposes one must restrict recursion to μ_*, that is, to type-level monomorphic recursion. In any case, the main system $F_\omega^{\mu_*;}$ is a non-trivial extension of (the contractive fragment of) $F_\omega^{\mu_*}$ (studied by Cai et al. [14]) as well as $F^{\mu;}$ (studied by Almeida et al. [19]).

We have only considered polymorphic types of a functional nature: type $\forall\alpha\colon\kappa.T$ must always be of kind $\mathtt{T}$. It is worth investigating polymorphism over session types, as it would allow further additional behaviour. For example, we could be interested in streaming values of heterogeneous nature, as in type $\mu\,\alpha\colon\mathtt{S}.\,\&\{\mathsf{Done}\colon\mathsf{Skip},\mathsf{More}\colon\forall\beta\colon\mathtt{T}.\,?\beta;\alpha\}$. It is however unclear whether this extension would still allow a translation into a simple grammar.

We proved that the type equivalence problem for systems F_ω^{μ}, $F_\omega^{\mu\cdot}$, $F_\omega^{\mu;}$ is at least as hard as a non-efficiently-decidable problem. We conjecture that these systems have the same power as deterministic pushdown automata (and hence, admit decidable type equivalence), but we do not have a construction to prove this result. In any case, our proof that the type equivalence problem is at least as hard as the bisimilarity of deterministic pushdown automata is enough to justify focus on the significant fragment with restricted recursion.

We study either full recursion (for theoretical results) or recursion limited to kind $*$ (for algorithmic results). It would be interesting to study in-between kinds of recursion; the next natural example is $\mu_{*\Rightarrow *}$. What model of computation would we arrive at if we consider types written with this recursion operator? We conjecture that types F_ω^{μ} and $F_\omega^{\mu\cdot}$, when restricted to recursion of kind $*\Rightarrow *$, would still be expressible as simple grammars, whereas such a restriction in the more powerful $F_\omega^{\mu;}$ would take us beyond this model, but perhaps without reaching the expressivity of deterministic pushdown automata.

References

1. Abel, A.: Type-based termination: a polymorphic lambda-calculus with sized higher-order types. Ph.D. thesis, Ludwig Maximilians University Munich (2007), https://d-nb.info/984765581
2. Almeida, B., Mordido, A., Thiemann, P., Vasconcelos, V.T.: Polymorphic lambda calculus with context-free session types. Inf. Comput. **289**(Part), 104948 (2022). https://doi.org/10.1016/j.ic.2022.104948
3. Almeida, B., Mordido, A., Vasconcelos, V.T.: FreeST: Context-free session types in a functional language. In: PLACES. EPTCS, vol. 291, pp. 12–23 (2019). https://doi.org/10.4204/EPTCS.291.2
4. Almeida, B., Mordido, A., Vasconcelos, V.T.: Deciding the bisimilarity of context-free session types. In: TACAS. LNCS, vol. 12079, pp. 39–56. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_3
5. Amadio, R.M., Cardelli, L.: Subtyping recursive types. In: POPL. pp. 104–118. ACM Press (1991). https://doi.org/10.1145/99583.99600
6. Baeten, J.C.M., Bergstra, J.A., Klop, J.W.: Decidability of bisimulation equivalence for processes generating context-free languages. In: PARLE. LNCS, vol. 259, pp. 94–111. Springer (1987). https://doi.org/10.1007/3-540-17945-3_5
7. Baeten, J.C.M., Bergstra, J.A., Klop, J.W.: Decidability of bisimulation equivalence for processes generating context-free languages. J. ACM **40**(3), 653–682 (1993). https://doi.org/10.1145/174130.174141
8. Balzer, S., Pfenning, F.: Manifest sharing with session types. Proc. ACM Program. Lang. **1**(ICFP), 37:1–37:29 (2017). https://doi.org/10.1145/3110281
9. Barendregt, H.P.: The lambda calculus - its syntax and semantics, Studies in logic and the foundations of mathematics, vol. 103. North-Holland (1985)
10. Barendregt, H.P.: The type free lambda calculus. In: Studies in Logic and the Foundations of Mathematics, vol. 90, pp. 1091–1132. Elsevier (1977)
11. Brandt, M., Henglein, F.: Coinductive axiomatization of recursive type equality and subtyping. Fundam. Informaticae **33**(4), 309–338 (1998). https://doi.org/10.3233/FI-1998-33401
12. Bruce, K.B., Cardelli, L., Pierce, B.C.: Comparing object encodings. In: TACS. LNCS, vol. 1281, pp. 415–438. Springer (1997). https://doi.org/10.1007/BFb0014561
13. Burkart, O., Caucal, D., Steffen, B.: An elementary bisimulation decision procedure for arbitrary context-free processes. In: MFCS. LNCS, vol. 969, pp. 423–433. Springer (1995). https://doi.org/10.1007/3-540-60246-1_148
14. Cai, Y., Giarrusso, P.G., Ostermann, K.: System F-omega with equirecursive types for datatype-generic programming. In: POPL. pp. 30–43. ACM (2016). https://doi.org/10.1145/2837614.2837660
15. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Behavioral polymorphism and parametricity in session-based communication. In: ESOP. LNCS, vol. 7792, pp. 330–349. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_19
16. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. ACM Comput. Surv. **17**(4), 471–522 (1985). https://doi.org/10.1145/6041.6042
17. Christensen, S., Hüttel, H., Stirling, C.: Bisimulation equivalence is decidable for all context-free processes. Inf. Comput. **121**(2), 143–148 (1995). https://doi.org/10.1006/inco.1995.1129

18. Colazzo, D., Ghelli, G.: Subtyping recursive types in kernel Fun. In: LICS. pp. 137–146. IEEE Computer Society (1999). https://doi.org/10.1109/LICS.1999.782605
19. Costa, D., Mordido, A., Poças, D., Vasconcelos, V.T.: Higher-order context-free session types in system F. In: PLACES. EPTCS, vol. 356, pp. 24–35 (2022). https://doi.org/10.4204/EPTCS.356.3
20. Costa, D., Mordido, A., Poças, D., Vasconcelos, V.T.: System F^{μ}_{ω} with context-free session types. CoRR **abs/2301.08659** (2023), http://arxiv.org/abs/2301.08659
21. Curry, H.H., Feys, R., Craig, W. (eds.): Combinatory Logic, Volume I. North-Holland (1958)
22. Dardha, O.: Recursive session types revisited. In: BEAT. EPTCS, vol. 162, pp. 27–34 (2014). https://doi.org/10.4204/EPTCS.162.4
23. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. Inf. Comput. **256**, 253–286 (2017). https://doi.org/10.1016/j.ic.2017.06.002
24. Das, A., DeYoung, H., Mordido, A., Pfenning, F.: Nested session types. In: ESOP. LNCS, vol. 12648, pp. 178–206. Springer (2021). https://doi.org/10.1007/978-3-030-72019-3_7
25. Das, A., DeYoung, H., Mordido, A., Pfenning, F.: Nested session types. ACM Trans. Program. Lang. Syst. **44**(3), 19:1–19:45 (2022). https://doi.org/10.1145/3539656
26. Das, A., Pfenning, F.: Rast: A language for resource-aware session types. Log. Methods Comput. Sci. **18**(1) (2022). https://doi.org/10.46298/lmcs-18(1:9)2022
27. De Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In: Indagationes Mathematicae. vol. 75, pp. 381–392. Elsevier (1972). https://doi.org/10.1016/1385-7258(72)90034-0
28. The FreeST programming language. https://freest-lang.github.io/ (2019)
29. Gapeyev, V., Levin, M.Y., Pierce, B.C.: Recursive subtyping revealed: functional pearl. In: ICFP. pp. 221–231. ACM (2000). https://doi.org/10.1145/351240.351261
30. Gauthier, N., Pottier, F.: Numbering matters: first-order canonical forms for second-order recursive types. In: ICFP. pp. 150–161. ACM (2004). https://doi.org/10.1145/1016850.1016872
31. Gay, S.J.: Bounded polymorphism in session types. MSCS **18**(5), 895–930 (2008). https://doi.org/10.1017/S0960129508006944
32. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Informatica **42**(2-3), 191–225 (2005). https://doi.org/10.1007/s00236-005-0177-z
33. Gay, S.J., Poças, D., Vasconcelos, V.T.: The different shades of infinite session types. In: FoSSaCS. LNCS, vol. 13242, pp. 347–367. Springer (2022). https://doi.org/10.1007/978-3-030-99253-8_18
34. Gay, S.J., Thiemann, P., Vasconcelos, V.T.: Duality of session types: The final cut. In: PLACES. EPTCS, vol. 314, pp. 23–33 (2020). https://doi.org/10.4204/EPTCS.314.3
35. Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. J. Funct. Program. **20**(1), 19–50 (2010). https://doi.org/10.1017/S0956796809990268
36. Girard, J.Y.: Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. Ph.D. thesis, Éditeur inconnu (1972)
37. Girard, J.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987). https://doi.org/10.1016/0304-3975(87)90045-4
38. Glew, N.: A theory of second-order trees. In: ESOP. LNCS, vol. 2305, pp. 147–161. Springer (2002). https://doi.org/10.1007/3-540-45927-8_11
39. Griffith, D.E.: Polarized substructural session types. Ph.D. thesis, University of Illinois at Urbana-Champaign (2016). https://doi.org/10.2172/1562827

40. Hindley, J.R., Seldin, J.P.: Introduction to Combinators and Lambda-Calculus. Cambridge University Press (1986)
41. Hinze, R.: Polytypic values possess polykinded types. Sci. Comput. Program. **43**(2-3), 129–159 (2002). https://doi.org/10.1016/S0167-6423(02)00025-4
42. Honda, K.: Types for dyadic interaction. In: CONCUR. LNCS, vol. 715, pp. 509–523. Springer (1993). https://doi.org/10.1007/3-540-57208-2_35
43. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP. LNCS, vol. 1381, pp. 122–138. Springer (1998). https://doi.org/10.1007/BFb0053567
44. Hopcroft, J.E., Karp, R.M.: A linear algorithm for testing equivalence of finite automata. Tech. rep., Cornell University (1971)
45. Im, H., Nakata, K., Park, S.: Contractive signatures with recursive types, type parameters, and abstract types. In: ICALP. LNCS, vol. 7966, pp. 299–311. Springer (2013). https://doi.org/10.1007/978-3-642-39212-2_28
46. Jančar, P.: Short decidability proof for DPDA language equivalence via 1st order grammar bisimilarity. CoRR **abs/1010.4760** (2010), http://arxiv.org/abs/1010.4760
47. Jančar, P.: Bisimilarity on basic process algebra is in 2-ExpTime (an explicit proof). Log. Methods Comput. Sci. **9**(1) (2012). https://doi.org/10.2168/LMCS-9(1:10)2013
48. Keizer, A.C., Basold, H., Pérez, J.A.: Session coalgebras: A coalgebraic view on session types and communication protocols. In: ESOP. LNCS, vol. 12648, pp. 375–403. Springer (2021). https://doi.org/10.1007/978-3-030-72019-3_14
49. Kiefer, S.: BPA bisimilarity is EXPTIME-hard. Inf. Process. Lett. **113**(4), 101–106 (2013). https://doi.org/10.1016/j.ipl.2012.12.004
50. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Trans. Program. Lang. Syst. **21**(5), 914–947 (1999). https://doi.org/10.1145/330249.330251
51. Korenjak, A.J., Hopcroft, J.E.: Simple deterministic languages. In: SWAT. pp. 36–46. IEEE Computer Society (1966). https://doi.org/10.1109/SWAT.1966.22
52. Kozen, D., Silva, A.: Practical coinduction. Math. Struct. Comput. Sci. **27**(7), 1132–1152 (2017). https://doi.org/10.1017/S0960129515000493
53. Lange, J., Yoshida, N.: Characteristic formulae for session types. In: TACAS. LNCS, vol. 9636, pp. 833–850. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_52
54. Lindley, S., Morris, J.G.: Talking bananas: structural recursion for session types. In: ICFP. pp. 434–447. ACM (2016). https://doi.org/10.1145/2951913.2951921
55. Milner, R.: Functions as processes. Math. Struct. Comput. Sci. **2**(2), 119–141 (1992). https://doi.org/10.1017/S0960129500001407
56. Padovani, L.: Context-free session type inference. ACM Trans. Program. Lang. Syst. **41**(2), 9:1–9:37 (2019). https://doi.org/10.1145/3229062
57. Pierce, B.C.: Types and programming languages. MIT Press (2002)
58. Puntigam, F.: Non-regular process types. In: Euro-Par. LNCS, vol. 1685, pp. 1334–1343. Springer (1999). https://doi.org/10.1007/3-540-48311-X_189
59. Ravara, A., Vasconcelos, V.T.: Behavioural types for a calculus of concurrent objects. In: Euro-Par. LNCS, vol. 1300, pp. 554–561. Springer (1997). https://doi.org/10.1007/BFb0002782
60. Reynolds, J.C.: Towards a theory of type structure. In: Programming Symposium. LNCS, vol. 19, pp. 408–423. Springer (1974). https://doi.org/10.1007/3-540-06859-7_148

61. Sénizergues, G.: The equivalence problem for deterministic pushdown automata is decidable. In: ICALP. LNCS, vol. 1256, pp. 671–681. Springer (1997). https://doi.org/10.1007/3-540-63165-8_221
62. Sénizergues, G.: L(A) = L(B)? decidability results from complete formal systems. In: ICALP. LNCS, vol. 2380, p. 37. Springer (2002). https://doi.org/10.1007/3-540-45465-9_4
63. Solomon, M.H.: Type definitions with parameters. In: POPL. pp. 31–38. ACM Press (1978). https://doi.org/10.1145/512760.512765
64. Stirling, C.: Decidability of DPDA equivalence. Theor. Comput. Sci. **255**(1-2), 1–31 (2001). https://doi.org/10.1016/S0304-3975(00)00389-3
65. Stirling, C.: Deciding DPDA equivalence is primitive recursive. In: ICALP. Lecture Notes in Computer Science, vol. 2380, pp. 821–832. Springer (2002). https://doi.org/10.1007/3-540-45465-9_70
66. Südholt, M.: A model of components with non-regular protocols. In: SC. LNCS, vol. 3628, pp. 99–113. Springer (2005). https://doi.org/10.1007/11550679_8
67. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: PARLE. LNCS, vol. 817, pp. 398–413. Springer (1994). https://doi.org/10.1007/3-540-58184-7_118
68. Thiemann, P., Vasconcelos, V.T.: Context-free session types. In: ICFP. pp. 462–475. ACM (2016). https://doi.org/10.1145/2951913.2951926
69. Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: PPDP. pp. 161–172. ACM (2011). https://doi.org/10.1145/2003476.2003499
70. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: ESOP. LNCS, vol. 7792, pp. 350–369. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_20
71. Toninho, B., Yoshida, N.: On polymorphic sessions and functions: A tale of two (fully abstract) encodings. ACM Trans. Program. Lang. Syst. **43**(2), 7:1–7:55 (2021). https://doi.org/10.1145/3457884
72. Tov, J.A.: Practical programming with substructural types. Ph.D. thesis, Northeastern University (2012)
73. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012). https://doi.org/10.1016/j.ic.2012.05.002
74. Walker, D.: Advanced Topics in Types and Programming Languages, chap. Substructural Type Systems, pp. 3–44. The MIT Press (2005)

Safe Session-Based Concurrency with Shared Linear State

Pedro Rocha(✉) and Luís Caires

NOVA LINCS, NOVA University of Lisbon, Portugal
`pms.rocha@campus.fct.unl.pt` `lcaires@fct.unl.pt`

We introduce CLASS, a session-typed, higher-order, core language that supports concurrent computation with shared linear state. We believe that CLASS is the first proposal for a foundational language able to flexibly express realistic concurrent programming idioms, with a type system ensuring all the following three key properties: CLASS programs never misuse or leak stateful resources or memory, they never deadlock, and they always terminate. CLASS owes these strong properties to a propositions-as-types foundation based on Linear Logic, which we conservatively extend with logically motivated constructs for shareable affine state. We illustrate CLASS expressiveness with several examples involving memory-efficient linked data structures, sharing of resources with linear usage protocols, and sophisticated thread synchronisation, which may be type-checked with a perhaps surprisingly light type annotation burden.

1 Introduction

Stateful programming involving concurrency and shared state plays a prominent role in modern software development, but, in practice, getting concurrent code right is still quite hard for common developers. Typical sources of "bugs" include resource leaks (forgetting to release unused memory or close a socket), violation of resource state preconditions (writing to a closed file or sending out-of-order messages), races (data invariant breaking, erratic sharing of resources), deadlocks (indefinite wait for lock release or incoming messages), livelocks, and even general non-termination. Fifty years ago Hoare noted [40]: "Parallel programs are particularly prone to time-dependent errors, which either cannot be detected by program testing nor by run-time checks. It is therefore very important that a high-level language designed for this purpose should provide complete security against time-dependent errors by means of a *compile-time* check". It does not come as a surprise that finding ways to approximate such certainly very ambitious goal is still today the object of exciting intense research.

In this paper, we approach this challenge by leveraging the propositions-as-types (PaT) paradigm towards the realm of concurrency and shared state. PaT is known to offer a unifying framework connecting logic, computation, and programming languages. Since the seminal work of Curry and Howard [42], it is a prolific structuring concept for designing and reasoning about programming languages (see [82]). Remarkably, languages derived within PaT intrinsically satisfy crucial properties: *type preservation* (since reduction corresponds to cut-reduction), *confluence* (since computation corresponds to proof simplification),

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 421–450, 2023.
https://doi.org/10.1007/978-3-031-30044-8_16

deadlock freedom (as a consequence of cut-elimination) and *livelock freedom / termination* (as a consequence of strong normalisation).

Although PaT has a traditional focus on functional computation, the emergence of linear logic has progressively motivated interpretations of stateful/resourceful computation [78,1,14,2,12], eventually leading to the discovery of tight correspondences between session types and linear logic [22,27,81]. These systems already capture aspects of state change, namely in the sequential execution of session protocols, thus raising the question of whether such approaches could be extended to express notions of shared mutable state, subject to interference, as found in typical imperative and concurrent programs. Recently, such challenge was addressed by several works [9,64,67]. In particular, [67] developed a first basic shared state model enjoying all the aforementioned strong properties of PaT. However, although [67] supports higher-order shareable store for pure values of replicated type, it forbids linear objects, such as stateful processes or data structures with update in-place, to be stored and shared as in languages like Java, Rust, and in the CLASS core language we introduce herein.

In this work, we develop a novel, more fundamental approach to shared state and PaT, and introduce CLASS, a typed, higher-order, session based core language that supports general concurrent computation with dynamically allocated shared linear (more precisely, affine) state. We believe that CLASS is the first proposal for a foundational language. able to flexibly express realistic concurrent programming idioms, while ensuring all the following three key properties by static typing: CLASS programs never misuse or leak stateful resources or memory, they never deadlock, and they always terminate.

Despite the strength of its type system, CLASS expressiveness and effectiveness substantially overcomes limitations of related works, as we show with compelling program examples that can be algorithmically typed for memory safety, dead- and live-lock freedom with a perhaps surprisingly light type annotation burden. CLASS owes these strong properties to is PaT foundation based on Second-Order Linear Logic, already known to capture the polymorphic session calculus and the linear System F [74], but which we conservatively extend with novel logically motivated constructs for shareable affine state, also based on DiLL co-exponentials [35,67], but to which we give here a different, more general and fundamental interpretation.

1.1 Overview

A main novelty and source of CLASS's expressiveness, flexibility and strong meta-theoretical properties resides in its mechanism for shared state composition. It is interesting to overview such mechanism in the context of the basic composition and interaction principles of the fundamental linear logic interpretations [22,27,81]. Our computational model is structured around processes that interact via binary sessions, the basic composition rules being mix and cut.

$$\frac{P \vdash \Delta_1; \Gamma \quad Q \vdash \Delta_2; \Gamma}{P \parallel Q \vdash \Delta_1, \Delta_2; \Gamma}\,[\text{Tmix}] \qquad \frac{P \vdash \Delta_1, x : A; \Gamma \quad Q \vdash \Delta_2, x : \overline{A}; \Gamma}{P \;|x|\; Q \vdash \Delta_1, \Delta_2; \Gamma}\,[\text{Tcut}]$$

The mix rule types the independent composition of processes P and Q, which do not share any free names and run side-by-side without interacting. This is captured by the implicit disjointness of their linear typing contexts Δ_1 and Δ_2, declaring the types of their interaction channels. Interactive composition is expressed by the cut rule, which connects exactly two processes P and Q through a *single* linear session x with *dual typed* endpoints ($x : A$ and $x : \overline{A}$), following Abramsky's idea of "cut as interactive composition" [1].

Intuitively, duality of endpoint (session) types ensures that all interactions between P and Q on x always matches: when P sends, Q receives; when Q offers, P chooses; and likewise for all types. Notice that sharing a single channel x between the threads P and Q is important to ensure acyclicity of proof structures, and cut-elimination/deadlock absence. But P, Q may use an arbitrary number of linear channels, in Δ_1, Δ_2, to also compose with other processes.

Shared composition in session types is available for *replicated* "server" objects $!x(y); P$, typed by the linear logic exponential type bang $!A$. Contraction of the dual exponential type why-not $?\overline{A}$ allows an unbounded number of usages of such replicated server object to be introduced in client processes. In the dyadic presentation of linear logic (cf. [5,11]), contraction is expressed by moving ?-typed names into the unrestricted context Γ, with the [T?] rule.

$$\dfrac{!x(y); P \vdash x :!A; \Gamma \qquad \dfrac{Q \vdash \Delta; \Gamma, x : \overline{A}}{?x; Q \vdash \Delta, x :?\overline{A}; \Gamma}\,[\mathrm{T}?]}{!x(y); P \;|x|\; ?x; Q \vdash \Delta; \Gamma} \qquad \dfrac{\dfrac{\vdots}{R \vdash \Delta, y : \overline{A}; \Gamma, x{:}\overline{A}}}{\mathsf{call}\ x(y); R \vdash \Delta; \Gamma, x{:}\overline{A}}\,[\mathrm{Tcall}]$$

Names in Γ may be used unrestrictedly; each call (typed by [Tcall]) spawns a fresh copy of the server body at type $y : A$, to be used by the client at type $y : \overline{A}$, in a linear binary session. By the typing rule for !A (promotion) such copy does not depend on linear resources. Thus, interaction with replicated objects as captured by the exponentials $!A$ and $?A$ implements a copy semantics where each call obtains a new private *stateless* copy of the same object.

In this work, we introduce a third composition mechanism, allowing processes to concurrently share mutex memory cells, storing *linear state*. Mutex memory cells and their usages are typed respectively by a pair of dual modalities $\mathsf{S}_\bullet A$ and $\mathsf{U}_\bullet A$, whose logical rules are motivated by Differential Linear Logic (DiLL) [35], in particular *cocontraction*, expressed by the type rule [Tsh].

$$\dfrac{P \vdash \Delta, x : \mathsf{U}_\bullet A; \Gamma \qquad Q \vdash \Delta', x : \mathsf{U}_\bullet A; \Gamma}{\mathsf{share}\ x\ \{P \parallel Q\} \vdash \Delta, \Delta', x : \mathsf{U}_\bullet A; \Gamma}\,[\mathrm{Tsh}]$$

While sharing of replicated objects corresponds to contraction of $?A$ types, shared usage of mutex cells corresponds to cocontraction of $\mathsf{U}_\bullet A$ types. Apart from the explicit use of [Tsh], the type system ensures that memory cells are always used linearly. The shared usage $x : \mathsf{U}_\bullet A$ *is free* in the conclusion of the typing rule, therefore a memory cell may be shared by an arbitrary number of processes, by nested iterated use of cocontraction.

Moreover, cocontraction also ensures that concurrent processes may share a single mutex cell (just like [Tcut] w.r.t. binary sessions). This constraint comes from the linear logic discipline, and it is important to ensure deadlock freedom. As discussed in Concluding Remarks, this does not hinder CLASS expressiveness - e.g., a single mutex cell may act as a gateway to further bundles of shared state, organised in resource hierarchies, as our examples illustrate - and even suggests convenient concurrent programming structuring techniques.

To access a mutex memory cell in its (unlocked) full state, typed by $\mathsf{U}_\bullet A$, the client uses a *take* operation. Take waits for acquiring the cell lock and reads its contents. The cell then transitions to the (locked) empty state, typed by $\mathsf{U}_\circ A$. The taking client becomes the sole responsible for filling back the cell contents, using a *put* operation. This will restore the cell to the full state, releasing its lock, and making it accessible to other concurrent threads waiting to take it. Our mutex memory cell object is thus akin to a behaviourally typed incarnation of Concurrent Haskell MVars [45] or Rust std::sync::Mutex objects [46].

To ensure safe releasing of a memory cell, its contents are required to be of affine type $\wedge A$. Affine objects are well-behaved disposable values, that when discarded, safely dispose all resources they hereditarily refer to, this being ensured by the linear logic typing.

We illustrate the introduced concepts with a simple example, where two concurrent threads compete to set *on* an initially *off* flag, but only one may win. The flag iteratively announces its state to the client with either #Off or #On. If the state is *off*, the client must select #turnOn, if the state is *on*, it will remain *on*. Process $\mathsf{flag}(f)$ implements the flag (at name f) in the *off* state, and process $\mathsf{on}(f)$ in the *on* state, defined thus

$$\begin{aligned} \mathsf{flag}(f) &= \#\mathsf{Off}\ f; \mathsf{case}\ f\{\ |\ \#\mathsf{turnOn} : \mathsf{affine}\ f; \mathsf{on}(f)\ \} \\ \mathsf{on}(f) &= \#\mathsf{On}\ f; \mathsf{affine}\ f; \mathsf{on}(f) \end{aligned}$$

The flag object is typed with the (linear) usage protocol defined by the coinductive type Flag below, such that $\mathsf{flag}(f) \vdash f : \mathsf{Flag}$ and $\mathsf{on}(f) \vdash f : \mathsf{Flag}$

$$\mathsf{type\ corec\ Flag} = \oplus\{\ |\#\mathsf{Off} : \&\{\ |\#\mathsf{turnOn} : \wedge\mathsf{Flag}\},\ \ |\#\mathsf{On} : \wedge\mathsf{Flag}\}$$

We now consider a scenario where a flag object is shared via a mutex memory cell c initially storing a *off* flag of type $\wedge\mathsf{Flag}$ among two concurrent clients.

```
$\mathsf{client}(c, \mathsf{id}) \vdash c : \mathsf{U}_\bullet\overline{\mathsf{Flag}}; \mathsf{id} : \overline{\mathsf{int}}$
client(c, id) =
   take c(f);
   case f {
      |#Off : println id + “: wins.’;
               #turnOn f;
               put c(f); release c
      |#On : println id + “: loses.’;
               put c(f); release c
   }
```

```
$\mathsf{main}() \vdash \emptyset$
main() =
   cut {  cell c(f.affine f; flag(f))
          |c : $\mathsf{U}_\bullet\overline{\mathsf{Flag}}$|
          share c {
                client(c, 1)
                ||
                client(c, 2)
          }
   }
```

When running main() exactly one of the threads (executing the same code, just with a different id) will turn the flag *on* and win, the other will loose. Notice that all threads drop usage of the memory cell c using release, which corresponds to DiLL coweakening ([35]).

When considering a new language, in particular with a static typing discipline, it is necessary to argue about its expressiveness, and aim for a better perception of how natural programs get past its typing rules, and of how types help in structuring programs. In this paper, we approach these concerns by showcasing many interesting examples that challenge the expressiveness of the CLASS language and type system on realistic concurrent programming scenarios. We have developed many more examples, distributed with our implementation [68], combining imperative, higher-order functional, and session-based programming styles. For all these programs, strong guarantees of memory safety, deadlock-freedom, termination, and absence of "dynamic bugs", even in the presence of blocking primitives and higher-order state, are compositionally certified by our lightweight type discipline based on Propositions-as-Types and Linear Logic.

1.2 Outline and Contributions

We believe that CLASS is the first proposal for a foundational language able to flexibly express realistic concurrent programming idioms while ensuring by typing three key properties: CLASS programs never misuse or leak stateful resources or memory, they never deadlock, and they always terminate.

In Section 2 we formally present the core language CLASS, its type system and operational semantics. Our model builds on the propositions-as-types approach to session-based concurrency [22,27,80], extending Second-Order Classical Linear Logic with inductive/coinductive types, affine types, and novel primitives for shareable first-class mutex reference cells for linear state.

In Section 3 we state and prove type preservation (Theorem 1), progress (Theorem 2) which implies deadlock-freedom, and strong normalisation (Theorem 3), which also implies livelock absence. Our proof uses a logical relations argument, extended with an interesting technique to handle shared state interference, which we believe is exploited here for the first time.

Given the strong properties of its type system, it is of course very important to substantiate our claims about CLASS expressiveness. In Section 4 we illustrate the expressiveness of CLASS language and type system by going through a series of compelling examples. Namely, we discuss a general technique for sharing linear protocols, a shareable linked list with update in-place, a shareable buffered channel, using a linked list with pointers to tail and head nodes, and executing send and receive operations in O(1) time; the dining philosophers, illustrating techniques that rely on our type structure to encode resource acquisition hierarchies; a generic barrier for n threads; and a Hoare style monitor with await/notify conditions, where our implementation of the condition's process queue is supported by a dynamic linked data structure, as in real systems code.

Section 5 discusses related work. Section 6 offers concluding remarks and suggests further research. Complete definitions and detailed proofs to all results are provided in [65].

2 The Core Language and its Type System

We present the core language, type system, and operational semantics of CLASS. The language is based on a PaT correspondence with Linear Logic, so terms of the language correspond to proof rules. We start by types and duality.

Definition 1 (Types). *Types A, B of* CLASS *are defined by*

$$\begin{array}{llllllll} A, B ::= X & \mid \mathbf{1} & \mid \bot & \mid A \,\&\, B & \mid A \oplus B & \mid A \mathbin{⅋} B & \mid A \otimes B \\ & \mid\, !A & \mid ?A & \mid \exists X.A \mid & \mid \forall X.A & \mid \mu X.\ A & \mid \nu X.\ A \\ & \mid \wedge A & \mid \vee A & \mid \mathsf{S}_{\bullet} A & \mid \mathsf{S}_{\circ} A & \mid \mathsf{U}_{\bullet} A & \mid \mathsf{U}_{\circ} A \end{array}$$

Types in the first two rows correspond to Second-Order Classical Linear Logic, extended with inductive/coinductive types (μ, ν). Types comprise variables (X), units ($\mathbf{1}$, $\bot$), multiplicatives ($\otimes$, $⅋$), additives ($\oplus$, $\&$), exponentials ($!$, $?$) and quantifiers ($\exists$, $\forall$). The third row extends basic types with affine ($\wedge, \vee$) and new modalities ($\mathsf{S}_{\bullet}, \mathsf{U}_{\bullet}, \mathsf{S}_{\circ}, \mathsf{U}_{\circ}$) to type shared *affine state*. Duality is the involution operation $A \mapsto \overline{A}$ on types, corresponding to Linear Logic negation, defined by

$$\begin{array}{lll} \overline{\mathbf{1}} \;\;= \bot & \overline{A \otimes B} = \overline{A} \mathbin{⅋} \overline{B} & \overline{A \oplus B} = \overline{A} \,\&\, \overline{B} \\ \overline{!A} \;= ?\overline{B} & \overline{\exists X.A} \;\; = \forall X.\overline{A} & \overline{\mu X.\ A} = \nu X.\ \{\overline{X}/X\}(\overline{A}) \\ \overline{\wedge A} = \vee \overline{A} & \overline{\mathsf{S}_{\bullet} A} \;\;\; = \mathsf{U}_{\bullet} \overline{A} & \overline{\mathsf{S}_{\circ} A} \;\;\;\; = \mathsf{U}_{\circ} \overline{A} \end{array}$$

Duality captures symmetry in process interaction, as manifest in the cut rule. In our system, typing judgements have the form $P \vdash_{\eta} \Delta; \Gamma$. The typing context $\Delta; \Gamma$ is dyadic [4,15,63,22], where Δ is handled linearly and Γ is unrestricted; both Δ and Γ assign types to names. The index η is a finite map that holds coinduction hypothesis to type corecursive processes, as detailed later.

Definition 2. *The typing rules of* CLASS *are presented in Figs. 1 to 5.*

The type system corresponds, via propositions-as-types [22,27,80], to Second-Order Classical Linear Logic (Fig. 1) with inductive/coinductive types (Fig. 2), affinity (Fig. 3) and extended with constructs for shared mutable state (Figs. 4 - 5). The basic composition rules are [Tmix] and [Tcut], which correspond to mix and cut of Linear Logic, respectively. [Tmix] types a parallel composition $P \mid\mid Q$, where P and Q run in parallel without interfering. On the other hand, [Tcut] types linear interactive composition $P \;|x : A|\; Q$: processes P and Q run concurrently and communicate through a private linear session x, session endpoints being typed by dual types $A/\overline{A}$. When the cut type annotation does not play any role, we may omit it and write $P \;|x|\; Q$. In examples, for readability, we use cut $\{P \;|x|\; Q\}$ and par $\{P \mid\mid Q\}$ instead of $P \;|x|\; Q$ and $P \mid\mid Q$, respectively.

For the basic process constructs [22,27,80,19], $\otimes/⅋$ type send and receive, $\oplus/\&$ type choice and offer (in examples we use labelled choice) , $!/?$ type

$$\frac{}{\mathsf{0} \vdash_\eta \emptyset; \Gamma}\,[\mathrm{T}\mathsf{0}] \qquad \frac{P \vdash_\eta \Delta'; \Gamma \quad Q \vdash_\eta \Delta; \Gamma}{P \mathbin{||} Q \ \vdash_\eta \Delta', \Delta; \Gamma}\,[\mathrm{Tmix}]$$

$$\frac{}{\mathsf{fwd}\ x\ y\ \vdash_\eta x : \overline{A}, y : A; \Gamma}\,[\mathrm{Tfwd}] \quad \frac{P \vdash_\eta \Delta', x : A; \Gamma \quad Q \vdash_\eta \Delta, x : \overline{A}; \Gamma}{P \ |x : A|\ Q \ \vdash_\eta \Delta', \Delta; \Gamma}\,[\mathrm{Tcut}]$$

$$\frac{}{\mathsf{close}\ x \vdash_\eta x : \mathbf{1}; \Gamma}\,[\mathrm{T}\mathbf{1}] \qquad \frac{Q \vdash_\eta \Delta; \Gamma}{\mathsf{wait}\ x; Q \ \vdash_\eta \Delta, x : \bot; \Gamma}\,[\mathrm{T}\bot]$$

$$\frac{P_1 \vdash_\eta \Delta, x : A; \Gamma \quad P_2 \vdash_\eta \Delta, x : B; \Gamma}{\mathsf{case}\ x\ \{|\mathsf{inl} : P_1,\ |\mathsf{inr} : P_2\} \ \vdash_\eta \Delta, x : A \,\&\, B; \Gamma}\,[\mathrm{T}\&]$$

$$\frac{Q_1 \vdash_\eta \Delta', x : A; \Gamma}{x.\mathsf{inl}; Q_1 \ \vdash_\eta \Delta', x : A \oplus B; \Gamma}\,[\mathrm{T}\oplus_l] \quad \frac{Q_2 \vdash_\eta \Delta', x : B; \Gamma}{x.\mathsf{inr}; Q_2 \ \vdash_\eta \Delta', x : A \oplus B; \Gamma}\,[\mathrm{T}\oplus_r]$$

$$\frac{P_1 \vdash_\eta \Delta_1, y : A; \Gamma \quad P_2 \vdash_\eta \Delta_2, x : B; \Gamma}{\mathsf{send}\ x(y.P_1); P_2 \ \vdash_\eta \Delta_1, \Delta_2, x : A \otimes B; \Gamma}\,[\mathrm{T}\otimes]$$

$$\frac{Q \vdash_\eta \Delta, z : A, x : B; \Gamma}{\mathsf{recv}\ x(z); Q \vdash_\eta \Delta, x : A \parr B; \Gamma}\,[\mathrm{T}\parr]$$

$$\frac{P \vdash_\eta y : A; \Gamma}{!x(y); P \ \vdash_\eta x : !A; \Gamma}\,[\mathrm{T}!] \qquad \frac{Q \vdash_\eta \Delta; \Gamma, x : A}{?x; Q \ \vdash_\eta \Delta, x : ?A; \Gamma}\,[\mathrm{T}?]$$

$$\frac{P \vdash_\eta y : A; \Gamma \quad Q \vdash_\eta \Delta; \Gamma, x : \overline{A}}{y.P \ |!x : A|\ Q \ \vdash_\eta \Delta; \Gamma}\,[\mathrm{Tcut!}] \qquad \frac{Q \vdash_\eta \Delta, z : A; \Gamma, x : A}{\mathsf{call}\ x(z); Q \ \vdash_\eta \Delta; \Gamma, x : A}\,[\mathrm{Tcall}]$$

$$\frac{P \vdash_\eta \Delta, x : \{B/X\}A; \Gamma}{\mathsf{sendty}\ x(B); P \ \vdash_\eta \Delta, x : \exists X.A; \Gamma}\,[\mathrm{T}\exists] \quad \frac{Q \vdash_\eta \Delta, x : A; \Gamma}{\mathsf{recvty}\ x(X); Q \vdash_\eta \Delta, x : \forall X.A; \Gamma}\,[\mathrm{T}\forall]$$

Fig. 1: Typing Rules I: Second-Order CLL.

$$\frac{P \vdash_{\eta'} \Delta, z : A; \Gamma \quad \eta' = \eta, X(z, \boldsymbol{w}) \mapsto \Delta, z : Y; \Gamma}{\mathsf{corec}\ X(z, \boldsymbol{w}); P\ [x, \boldsymbol{y}] \vdash_\eta \{\boldsymbol{y}/\boldsymbol{w}\}\Delta, x : \nu Y.\ A; \{\boldsymbol{y}/\boldsymbol{w}\}\Gamma}\,[\mathrm{Tcorec}]$$

$$\frac{\eta = \eta', X(x, \boldsymbol{y}) \ \mapsto \Delta, x : Y; \Gamma}{X(z, \boldsymbol{w}) \vdash_\eta \{\boldsymbol{w}/\boldsymbol{y}\}\Delta, z : Y; \{\boldsymbol{w}/\boldsymbol{y}\}\Gamma}\,[\mathrm{Tvar}]$$

$$\frac{P \vdash_\eta \Delta, x : \{\mu X.\ A/X\}A; \Gamma}{\mathsf{unfold}_\mu\ x; P \vdash_\eta \Delta, x : \mu X.\ A; \Gamma}\,[\mathrm{T}\mu] \qquad \frac{P \vdash_\eta \Delta, x : \{\nu X.\ A/X\}A; \Gamma}{\mathsf{unfold}_\nu\ x; P \vdash_\eta \Delta, x : \nu X.\ A; \Gamma}\,[\mathrm{T}\nu]$$

Fig. 2: Typing Rules II: Induction and Coinduction.

$$\frac{P \vdash_\eta a : A, \boldsymbol{b} : \vee \boldsymbol{B}, \boldsymbol{c} : \mathsf{U}_\bullet \boldsymbol{C}; \Gamma}{\mathsf{affine}_{\boldsymbol{b},\boldsymbol{c}}\ a; P\ \vdash_\eta a : \wedge A, \boldsymbol{b} : \vee \boldsymbol{B}, \boldsymbol{c} : \mathsf{U}_\bullet \boldsymbol{C}; \Gamma}\ [\text{Taffine}]$$

$$\frac{}{\mathsf{discard}\ a \vdash_\eta a : \vee A; \Gamma}\ [\text{Tdiscard}] \qquad \frac{Q \vdash_\eta \Delta, a : A; \Gamma}{\mathsf{use}\ a; Q \vdash_\eta \Delta, a : \vee A; \Gamma}\ [\text{Tuse}]$$

Fig. 3: Typing Rules III: Affinity.

replicated servers and their invocation, $\forall/\exists$ type receive and send of types, implementing polymorphic processes.

Coinductive types are introduced by rule [Tcorec]. It types corecursive processes $\mathsf{corec}\ X(z, \boldsymbol{w}); P\ [x, \boldsymbol{y}]$, with parameters $z, \boldsymbol{w}$ bound in P, that are instantiated with the arguments $x, \boldsymbol{y}$ (free in the process term). By convention, the coinductive behaviour, of type $\nu Y.\ A$, of a corecursive process is always offered in the first argument z. According to [Tcorec], to type the body P of a corecursive process, the map η is extended with a coinductive hypothesis binding the process variable X to the typing context $\Delta, z : Y; \Gamma$, so that when typing the body P of the corecursion we can appeal to X, which intuitively stands for P itself, and recover its typing invariant. Crucially, the type variable Y is free only in $z : A$. This causes corecursive calls to be always applied to names z' that hereditarily descend from the initial corecursive argument z, a necessary condition for strong normalisation (Theorem 3), and morally corresponds to only allowing corecursive calls on "smaller" argument sessions (of inductive type).

Rule [Tvar] types a corecursive call $X(z, \boldsymbol{w})$ by looking up in η for the corresponding binding and renaming the parameters with the arguments of the call. Inductive and coinductive types are explicitly unfolded with [Tμ] and [Tν].

To simplify the presentation in program examples, we omit explicit unfolding actions, and write inductive and coinductive type definitions with equations of the form $\mathsf{rec}\ A = f(A)$ and $\mathsf{corec}\ B = f(B)$ instead of $A = \mu X.\ f(X)$ and $B = \nu X.\ f(X)$, respectively. Similarly, we write corecursive process definitions as $Q(x, \boldsymbol{y}) = f(Q(-))$ instead of $Q(x, \boldsymbol{y}) = \mathsf{corec}\ X(z, \boldsymbol{w}); f(X(-))\ [x, \boldsymbol{y}]$, while of course respecting the constraints imposed by typing rules [Tvar] and [Tcorec].

Affinity Affinity is important to model discardable linear resources, and plays an important role in CLASS. An affine session can either be used as a linear session or discarded. The typing rules for the affine modalities are in Fig. 3. Affine sessions are introduced by rule [Taffine] that promotes a linear $a : A$ to an affine session $a : \wedge A$. It types $\mathsf{affine}_{\boldsymbol{b},\boldsymbol{c}}\ a; P$, which provides an affine session at a and continues as P, and follows the structure of a standard promotion rule.

A session a may be promoted to affine if it only depends on resources that can be disposed, i.e. resources that satisfy some form of weakening capability, namely: coaffine sessions b_i of type $\vee B_i$, that can be discarded; full cell usages c_i of type with $\mathsf{U}_\bullet C_i$, that can be released; and unrestricted sessions in Γ, which are implicitly ?-typed. The dependencies of an affine object on coaffine or full

$$\frac{P \vdash_\eta \Delta, a : \wedge A; \Gamma}{\mathsf{cell}\ c(a.P) \vdash_\eta \Delta, c : \mathsf{S}_\bullet A; \Gamma}\ [\text{Tcell}] \qquad \frac{}{\mathsf{release}\ c \vdash_\eta c : \mathsf{U}_\bullet A; \Gamma}\ [\text{Trelease}]$$

$$\frac{}{\mathsf{empty}\ c \vdash_\eta c : \mathsf{S}_\circ A; \Gamma}\ [\text{Tempty}] \qquad \frac{Q \vdash_\eta \Delta, a : \vee A, c : \mathsf{U}_\circ A; \Gamma}{\mathsf{take}\ c(a); Q \vdash_\eta \Delta, c : \mathsf{U}_\bullet A; \Gamma}\ [\text{Ttake}]$$

$$\frac{Q_1 \vdash_\eta \Delta_1, a : \wedge \overline{A}; \Gamma \qquad Q_2 \vdash_\eta \Delta_2, c : \mathsf{U}_\bullet A; \Gamma}{\mathsf{put}\ c(a.Q_1); Q_2\ \vdash_\eta \Delta_1, \Delta_2, c : \mathsf{U}_\circ A; \Gamma}\ [\text{Tput}]$$

Fig. 4: Typing Rules IV: Reference Cells.

cell objects are explicitly annotated as $\boldsymbol{b}, \boldsymbol{c}$ in the process term, to instrument the operational semantics, but we often omit them and simply write affine $a; P$.

The coaffine endpoint $\vee A$ of an affine session, dual of $\wedge \overline{A}$, has two operations: use and discard. Rule [Tuse] types a process use $a; Q$ that uses a coaffine session a and continues as Q, it is a dereliction rule. [Tdiscard] types the process discard a that discards a coaffine session a, it is a weakening rule.

Shared Mutable State Shared state is introduced in CLASS by typed constructs that model mutex memory cells, and associated cell operations allowing its use by client code, defined by the tying rules in Fig. 4.

At any moment a cell may be either *full* or *empty*, akin to the MVars of Concurrent Haskell [45]. A full cell on c, written cell $c(a.P)$, is typed $\mathsf{S}_\bullet A$ by rule [Tcell]. Such cell stores an *affine* session of type $\wedge A$, implemented at a by P. All objects stored in cells are required to be affine, so that memory cells may always be safely disposed without causing memory leaks. An empty cell on c, of type $\mathsf{S}_\circ A$, and written empty c, is typed by rule [Tempty].

Client processes manipulate cells via *take*, *put* and *release* operations. These operations apply to names of cell usage types - $\mathsf{U}_\bullet A$ (full cell usage) and $\mathsf{U}_\circ A$ (empty cell usage) - which are dual types of $\mathsf{S}_\bullet \overline{A}$ and $\mathsf{S}_\circ \overline{A}$, respectively. At any given moment, a client thread owning a $\mathsf{U}_\bullet A$-typed usage to a cell may execute a *take* operation, typed by rule [Ttake]. The *take* operation take $c(a); Q$ waits to acquire the cell mutex c, and reads its contents into parameter a, the linear (actually coaffine, of type $\vee A$) usage for the object stored in the cell; the cell becomes empty, and execution continues as Q.

It is responsibility of the taking thread to put some value back in the empty cell, thus releasing the lock, causing the cell to transition to the full state. The *put* operation put $c(a.Q_1); Q_2$ is typed by [Tput], the stored object a, implemented by Q_1, is required to be affine, as specified in the premise $a : \wedge \overline{A}$.

Hence a cell flips from full to empty and back; [Ttake] uses the cell c at $\mathsf{U}_\bullet A$ type, and its continuation (in the premise) at $\mathsf{U}_\circ A$ type, symmetrically [Tput] uses the cell c at $\mathsf{U}_\circ A$ type, and its continuation (in the premise) at $\mathsf{U}_\bullet A$ type.

The release c operation allows a thread to manifestly drop its cell usage c. Release is typed by [Trelease] (cf. coweakening [35]); a usage may only be released

$$\frac{P \vdash_\eta \Delta', c : \mathsf{U}_\bullet A; \Gamma \quad Q \vdash_\eta \Delta, c : \mathsf{U}_\bullet A; \Gamma}{\mathsf{share}\ c\ \{P \mid\mid Q\} \ \vdash_\eta \Delta', \Delta, c : \mathsf{U}_\bullet A; \Gamma}\ [\text{Tsh}]$$

$$\frac{P \vdash_\eta \Delta', c : \mathsf{U}_\circ A; \Gamma \quad Q \vdash_\eta \Delta, c : \mathsf{U}_\bullet A; \Gamma}{\mathsf{share}\ c\ \{P \mid\mid Q\} \ \vdash_\eta \Delta', \Delta, c : \mathsf{U}_\circ A; \Gamma}\ [\text{TshL}]$$

$$\frac{P \vdash_\eta \Delta', c : \mathsf{U}_\bullet A; \Gamma \quad Q \vdash_\eta \Delta, c : \mathsf{U}_\circ A; \Gamma}{\mathsf{share}\ c\ \{P \mid\mid Q\} \ \vdash_\eta \Delta', \Delta, c : \mathsf{U}_\circ A; \Gamma}\ [\text{TshR}]$$

Fig. 5: Typing Rules V: State Sharing.

in the unlocked state $\mathsf{U}_\bullet A$. When, for some cell c, all the owning threads release their usages, which eventually happens in well-typed programs, the cell c gets disposed, and its (affine) contents safely discarded.

Our memory cells cells are linear objects, with a linear mutable payload, which are never duplicated by reduction or conversion rules. However, in CLASS, multiple cell usages may be shared between concurrent threads, which compete to take and use it in interleaved critical sections. Such aliased usages be passed around and duplicated dynamically, changing the sharing topology at runtime.

Sharing of cell usages is logically expressed in our system by the typing rules in Fig. 5. Co-contraction, introduced in Differential Linear Logic DiLL [35], allows finite multisets of linear resources to safely interact in cut-reduction, resolving concurrent sharing into nondeterminism, as required here to soundly model memory cells and their linear concurrent usages. Rule [Tsh] interprets cocontraction with the construct $\mathsf{share}\ c\ \{P \mid\mid Q\}$, and types sharing of the cell usage $c : \mathsf{U}_\bullet A$ between the concurrent threads P and Q.

Contrary to cut, $\mathsf{share}\ c\ \{P \mid\mid Q\}$ is *not* a binding operator for c. The shared usage $c : \mathsf{U}_\bullet A$ is *free* in the conclusion of the typing rule, permitting c to be shared among an arbitrary number of threads, by nested iterated use of [Tsh]. In [Tsh], P and Q only share the single mutex cell c, since the linear context is split multiplicatively, just like [Tcut] wrt. binary sessions. This condition comes from the DiLL typing discipline, and is important to ensure deadlock freedom.

While [Tsh] types sharing of a full (unlocked) cell usage of type $\mathsf{U}_\bullet A$, the symmetric rules [TshR] and [TshR] type sharing of an empty (locked) cell usage of type $\mathsf{U}_\circ A$. We may verify that for every cell c in a well-typed process, at most one unguarded operation to c may be using type $\mathsf{U}_\circ A$, all the remaining unguarded operations to c must be using type $\mathsf{U}_\bullet A$. This implies that, at runtime, only one thread may own the lock for a given (necessarily empty) cell, and execute a *put* to it, which will bring the cell back to full and release its lock, other threads must be either attempting to *take*, or *release* the reference.

Working together, the sharing typing rules ensure that in any well-typed cell sharing tree, at most one single thread at any time may be actively using a cell (in the locked empty state) and put to it, thus guaranteeing mutual exclusion, while satisfying Progress (Theorem 2) which in turn ensures deadlock absence, even in the presence of the crucially blocking behaviour of the take operation.

2.1 Operational Semantics

We now define CLASS operational semantics, which is given by a structural precongruence relation $\leq$ that captures static relations on processes, essentially rearranging them, and a reduction relation $\rightarrow$ that captures process interaction.

Definition 3 **($P \equiv Q$ and $P \leq Q$).** *Structural congruence $\equiv$ is the least congruence on processes closed under α-conversion and the $\equiv$-rules in Fig. 6. Structural precongruence $\leq$ is the least precongruence on processes including $\equiv$ and closed under α-conversion and the $\leq$-rules in Fig. 6.*

The basic rules of $\equiv$ essentially reflect the expected static laws, along the lines of the structural congruences / conversions in [22,80]. The binary operators forwarder, cut and share are commutative ([comm]). The set of processes modulo $\equiv$ is a commutative monoid with binary operation given by parallel composition and identity given by inaction $\mathsf{0}$ ([par]). Any two static constructs commute, as expressed by the laws [CM]-[ShC!]. Furthermore, we can distribute the unrestricted cut over all the static constructs as expressed by law [D-C!X], where $*$ stands for either a mix, linear or unrestricted cut or a share.

The commuting conversions [ShTake] and [ShPut] allows take and put operations on cell usages to commute with a share construct. Rule [ShTake] picks the take that occurs on the left argument, however since share is commutative, a right-biased version of [ShTake] is admissible. Using [ShTake], any of the two possible interleavings for two concurrent takes may be nondeterministically picked via $\leq$. Indeed, we express $\leq$ as a precongruence because it introduces nondeterminism, and does not express a behavioural equivalence as $\equiv$ does. N.B.: Although one could easily formulate a confluent version of CLASS semantics, using explicit sums as in [13,66,35,65], we prefer in this paper to focus on the expressiveness of CLASS as a programming language and on its deadlock and livelock absence properties, adopting a nondeterministic reduction relation.

In [ShPut] only a put, in the $\mathsf{U}_{\circ}A$-typed premise of [TshL], may be propagated up and eventually update the cell, causing it to transit back to the full state. Hence, take operations originating the $\mathsf{U}_{\bullet}A$ typed premise of [TshR] will be blocked, waiting until such (unique) put propagation occurs. Algebraically, rule [ShRel] expresses that the release operation is the identity for share composition, we orient it as a precongruence, to ensure type preservation.

Definition 4 **(Reduction $\rightarrow$).** *Reduction $\rightarrow$ is defined by the rules of Fig. 7.*

We let $\xrightarrow{*}$ stand for the reflexive-transitive closure of $\rightarrow$. Reduction includes the set of principal cut conversions, i.e. the redexes for each pair of interacting constructs. It is closed by structural precongruence ([$\leq$]) and in rule [cong] we consider that $\mathcal{C}$ is a static context, i.e. a process context in which the hole is covered only by the static constructs mix, cut and share.

$$
\begin{array}{ll}
\mathsf{fwd}\ x\ y \ \equiv \mathsf{fwd}\ y\ x \quad P\ |x|\ Q\ \equiv Q\ |x|\ P & \\
\mathsf{share}\ x\ \{P \mid\mid Q\} \equiv \mathsf{share}\ x\ \{Q \mid\mid P\} & [\text{comm}] \\
P \mid\mid \mathsf{0}\ \equiv P \quad P \mid\mid Q \equiv Q \mid\mid P \quad P \mid\mid (Q \mid\mid R) \equiv (P \mid\mid Q) \mid\mid R & [\text{par}] \\
P\ |x|\ (Q \mid\mid R) \equiv (P\ |x|\ Q) \mid\mid R & [\text{CM}] \\
P\ |x|\ (Q\ |y|\ R) \equiv (P\ |x|\ Q)\ |y|\ R & [\text{CC}] \\
P\ |x|\ \mathsf{share}\ y\ \{Q \mid\mid R\} \equiv \mathsf{share}\ y\ \{P\ |x|\ Q \mid\mid R\} & [\text{CSh}] \\
P\ |z|\ (y.Q\ |!x|\ R) \equiv y.Q\ |!x|\ (P\ |z|\ R) & [\text{CC!}] \\
y.Q\ |!x|\ (P \mid\mid R) \equiv P \mid\mid (y.Q\ |!x|\ R) & [\text{C!M}] \\
y.P\ |!x : A|\ (w.Q\ |!z : B|\ R) \equiv w.Q\ |!z : B|\ (y.P\ |!x : A|\ R) & [\text{C!C!}] \\
\mathsf{share}\ x\ \{P \mid\mid (Q \mid\mid R)\} \equiv \mathsf{share}\ x\ \{P \mid\mid Q\} \mid\mid R & [\text{ShM}] \\
\mathsf{share}\ x\ \{P \mid\mid \mathsf{share}\ y\ \{Q \mid\mid R\}\} \equiv \mathsf{share}\ y\ \{\mathsf{share}\ x\ \{P \mid\mid Q\} \mid\mid R\} & [\text{ShSh}] \\
\mathsf{share}\ z\ \{P \mid\mid y.Q\ |!x|\ R\} \equiv y.Q\ |!x|\ \mathsf{share}\ z\ \{P \mid\mid R\} & [\text{ShC!}] \\
y.P\ |!x : A|\ (Q * R) \equiv (y.P\ |!x : A|\ Q) * (y.P\ |!x : A|\ R) & [\text{D-C!X}] \\
\mathsf{share}\ x\ \{\mathsf{release}\ x \mid\mid P\} \leq P & [\text{ShRel}] \\
\mathsf{share}\ x\ \{\mathsf{put}\ x(y.P); Q \mid\mid R\} \leq \mathsf{put}\ x(y.P); \mathsf{share}\ x\ \{Q \mid\mid R\} & [\text{ShPut}] \\
\mathsf{share}\ x\ \{\mathsf{take}\ x(y_1); P_1 \mid\mid \mathsf{take}\ x(y_2); P_2\} & \\
\qquad \leq \mathsf{take}\ x(y_1); \mathsf{share}\ x\ \{P_1 \mid\mid \mathsf{take}\ x(y_2); P_2\} & [\text{ShTake}]
\end{array}
$$

Provisos: in [CM] and [ShM], $x \in \mathsf{fn}(Q)$; in [CC], [CSh] and [ShSh], $x, y \in \mathsf{fn}(Q)$; in [CC!], [C!M] and [ShC!], $x \notin \mathsf{fn}(P)$; in [C!C!], $x \notin \mathsf{fn}(Q)$ and $z \notin \mathsf{fn}(P)$.

Fig. 6: Structural congruence $P \equiv Q$ and precongruence $P \leq Q$.

Operationally, the forwarding behaviour is implemented by name substitution [23] ([fwd]). All the other conversions apply to a principal cut between two dual actions. Reduction rules for the basic session constructs that interpret Second Order Linear Logic and recursion are the expected ones [22,27,81], along predictable lines. For readability, we omit the type declarations in the cuts, as they do not actually play any role in reduction.

We comment the rules concerning affinity. The interaction between an affine session and an use operation is defined by reduction rule [$\wedge\vee$u], where a cut on $a : \wedge A$ between $\mathsf{affine}_{\boldsymbol{b},\boldsymbol{c}}\ a; P$ and $\mathsf{use}\ a; Q$ reduces to a cut on $a : A$ between the continuations P and Q. The reduction between an affine session and a discard operation is defined by [$\wedge\vee$d]. A cut between $\mathsf{affine}_{\boldsymbol{b},\boldsymbol{c}}\ a; P$ and $\mathsf{discard}\ a$ reduces to a mix-composition of discards (for the coaffine sessions $\boldsymbol{b}$) and releases (for the cell usages $\boldsymbol{c}$) cf. [6,20]). In the corner case where $\boldsymbol{c}$ and $\boldsymbol{a}$ are empty, the left-hand side of [$\wedge\vee$d] simply degenerates to inaction $\mathsf{0}$ (the identity of mix).

The reductions for the mutable state operations are fairly self-explanatory. In rule [$\mathsf{S}_\bullet\mathsf{U}_\bullet$r], a cut between a full mutex cell cell and a release operation reduces to a process that discards the affine cell contents, cf. rule [$\wedge\vee$d]. In rule [$\mathsf{S}_\bullet\mathsf{U}_\bullet$t], a cut on $c : \mathsf{S}_\bullet A$ between a full cell and a take operation reduces to a process with

$$\begin{array}{ll}
\mathsf{fwd}\ x\ y\ |y|\ P \to \{x/y\}P & [\mathrm{fwd}] \\
\mathsf{close}\ x\ |x|\ \mathsf{wait}\ x; P \to P & [\mathbf{1}\bot] \\
\mathsf{send}\ x(y.P); Q\ |x|\ \mathsf{recv}\ x(z); R \to Q\ |x|\ (P\ |y|\ \{y/z\}R) & [\otimes\parr] \\
\mathsf{case}\ x\ \{|\mathsf{inl} : P,\ |\mathsf{inr} : Q\}\ |x|\ x.\mathsf{inl}; R \to P\ |x|\ R & [\&\oplus_l] \\
\mathsf{case}\ x\ \{|\mathsf{inl} : P,\ |\mathsf{inr} : Q\}\ |x|\ x.\mathsf{inr}; R \to Q\ |x|\ R & [\&\oplus_r] \\
!x(y); P\ |x|\ ?x; Q \to y.P\ |!x|\ Q & [!?] \\
y.P\ |!x|\ \mathsf{call}\ x(z); Q \to \{z/y\}P\ |z|\ (y.P\ |!x|\ Q) & [\mathrm{call}] \\
\mathsf{sendty}\ x(A); P\ |x|\ \mathsf{recvty}\ x(X); Q \to P\ |x|\ \{A/X\}Q & [\exists\forall] \\
\mathsf{unfold}_\mu\ x; P\ |x|\ \mathsf{unfold}_\nu\ x; Q \to P\ |x|\ Q & [\mu\nu] \\
\mathsf{unfold}_\mu\ x; P\ |x|\ \mathsf{corec}\ Y(z, \boldsymbol{w}); Q\ [x, \boldsymbol{y}] & \\
\quad \to P\ |x|\ \{x/z\}\{\boldsymbol{y}/\boldsymbol{w}\}\{\mathsf{corec}\ Y(z, \boldsymbol{w}); Q/Y\}Q & [\mathrm{corec}] \\
\mathsf{affine}_{\boldsymbol{b},\boldsymbol{c}}\ a; P\ |a|\ \mathsf{use}\ a; Q \to P\ |a|\ Q & [\wedge\vee\mathrm{u}] \\
\mathsf{affine}_{\boldsymbol{b},\boldsymbol{c}}\ a; P\ |a|\ \mathsf{discard}\ a \to \mathsf{discard}\ \boldsymbol{b} \parallel \mathsf{release}\ \boldsymbol{c} & [\wedge\vee\mathrm{d}] \\
\mathsf{cell}\ c(a.P)\ |c|\ \mathsf{release}\ c \to P\ |a|\ \mathsf{discard}\ a & [\mathsf{S}_\bullet\mathsf{U}_\bullet\mathrm{r}] \\
\mathsf{cell}\ c(a.P)\ |c|\ \mathsf{take}\ c(a'); Q \to P\ |a|\ (\mathsf{empty}\ c\ |c|\ \{a/a'\}Q) & [\mathsf{S}_\bullet\mathsf{U}_\bullet\mathrm{t}] \\
\mathsf{empty}\ c\ |c|\ \mathsf{put}\ c(a.P); Q \to \mathsf{cell}\ c(a.P)\ |c|\ Q & [\mathsf{S}_\circ\mathsf{U}_\circ] \\
P \leq P' \text{ and } P' \to Q' \text{ and } Q' \leq Q\ \supset\ P \to Q & [\leq] \\
P \to Q\ \supset\ \mathcal{C}[P]\ \to \mathcal{C}[Q] & [\mathrm{cong}]
\end{array}$$

Fig. 7: Reduction $P \to Q$.

two cuts, both composed with the continuation $\{a/a'\}Q$ of the take. The outer cut on $a : \wedge A$ composes with the stored affine session, which was successfully acquired by the take operation. The inner cut on $c : \mathsf{S}_\circ A$ composes with the reference cell c, which has became empty in the reductum. Finally, in rule $[\mathsf{S}_\circ\mathsf{U}_\circ]$, a cut on session $c : \mathsf{S}_\circ A$ between an empty cell and a put operation reduces to a cut on session $c : \mathsf{S}_\bullet A$ between a full cell, that now stores the session that was put, and the continuation of the put process. Notice that the locking/unlocking behaviour of cells is simply modelled by rewriting of the process terms, from cell to empty and back, as typical in process calculi.

3 Type Safety and Strong Normalisation

In this section we state and give proof sketches for our main results of type safety and strong normalisation. Full proofs may be found in [65].

Type Preservation The semantics of CLASS is defined by a set of precongruence $\leq$ and reduction $\to$ rules on process terms. Theorem 1 shows that these relations preserve typing, and gives substance to our PaT approach, showing that every $\leq$ and $\to$ rule corresponds to a conversion on type derivations/proofs.

Theorem 1 (Type Preservation). *Suppose $P \vdash_\eta \Delta; \Gamma$. (1) If $P \leq Q$, then $Q \vdash_\eta \Delta; \Gamma$. (2) If $P \to Q$, then $Q \vdash_\eta \Delta; \Gamma$.*

Proof. By induction on derivations for $P \leq Q$ (resp. $P \to Q$), we verify that all the rules of $\leq$ (Def. 3) (resp. $\to$ (Def. 4)) are type preserving.

Progress We prove the progress property for well-typed CLASS processes. The following notion of *live process* becomes useful. A process P is *live* if and only if $P = \mathcal{C}[Q]$, for some static context $\mathcal{C}$ (the hole lies within the scope of static constructs mix, cut and share) and Q is an active process (a process with a topmost action prefix, such as a receive or a take, or a forwarder). We first show that a live well-typed process either reduces or offers an interaction with its environment on a free name. The following observability predicate (cf. [70]) characterises the interactions of a process with its environment

Definition 5 ($P \downarrow_x$). *The predicate $P \downarrow_x$ is defined by rules of Fig. 8.*

The predicate $P \downarrow_x$ holds if P offers an immediate interaction (unguarded action) on free name x. We can observe the subject of an action (rule [act]) and x, y of a forwarder fwd x y. The definition of $P \downarrow_x$ is closed by $\leq$ and propagates observations over the various static operators. Cut bound names are not free, hence cannot be observed. Share share y $\{P \mid\mid Q\}$ propagates all the observations x for which $x \neq y$ and by applying $\leq$ rules [ShTake], [ShRel] or [ShPut] via [$\leq$], an interaction on x may be observed. We have

Lemma 1 (Liveness). *Let $P \vdash_\emptyset \Delta; \Gamma$ be live. Either $P \downarrow_x$ or P reduces.*

Proof. (Sketch) By induction on a derivation for $P \vdash_\emptyset \Delta; \Gamma$, along the lines of [27]. To handle case [Tcut] $P = P_1 \; |y| \; P_2$: both P_1 and P_2 are live, since both type with a nonempty linear typing context, hence we can apply the induction hypothesis (i.h.) to both premises of [Tcut]: either (i) one of P_1 and P_2 reduces or (ii) both $P_1 \downarrow_{x_1}$ and $P_2 \downarrow_{x_2}$. If (i), then P reduces. Case (ii) follows because, crucially, P_1 and P_2 synchronise through a single private session y, then either $x_1 \neq y$ or $x_2 \neq y$, in which case we can observe either x_1 or x_2; or $x_1 = x_2 = y$, in which case we can trigger a reduction, by applying $\leq$ rules to P in order to exhibit a principal cut. For case [Tsh] $P =$ share y $\{P_1 \mid\mid P_2\}$: since P_1 and P_2 are live, we apply i.h. to both premises. The interesting case occurs when $P_1 \downarrow_{x_1}$ and $P_2 \downarrow_{x_2}$. Co-contraction implies that P_1 and P_2 share the single usage y, so if $x_1 \neq y$ or $x_2 \neq y$, we have either $P_1 \downarrow_{x_1}$ or $P_1 \downarrow_{x_2}$. If both $x_1 = x_2 = y$, then we derive $P \downarrow_y$: the observation corresponds to either a take or a release operation on y, which we commute up with [ShTake] or [ShRel]. For [TshL] $P =$ share y $\{P_1 \mid\mid P_2\}$, we apply the i.h. to the premise P_1, which types with an empty usage on y. If $P_1 \downarrow_y$, then $P \downarrow_y$, the observation corresponding a put operation on y, which we commute up with [ShPut]. Symmetrically for [TshR].

Theorem 2 (Progress). *Let $P \vdash_\emptyset \emptyset; \emptyset$ be a live process. Then, P reduces.*

Proof. Follows from Lemma 1 since $\mathsf{fn}(P) = \emptyset$.

$$\frac{}{\mathsf{fwd}\ x\ y \downarrow_x}[\text{fwd}] \quad \frac{s(\mathcal{A}) = x}{\mathcal{A} \downarrow_x}[\mathcal{A}] \quad \frac{P \leq Q \quad Q \downarrow_x}{P \downarrow_x}[\leq] \quad \frac{P \downarrow_x}{(P \mid\mid Q) \downarrow_x}[\text{mix}]$$

$$\frac{P \downarrow_x \quad x \neq y}{(P \mid y \mid Q) \downarrow_x}[\text{cut}] \quad \frac{Q \downarrow_x \quad x \neq y}{(z.P \mid !y \mid Q) \downarrow_x}[\text{cut!}] \quad \frac{P \downarrow_x \quad x \neq y}{(\mathsf{share}\ y\ \{P \mid\mid Q\}) \downarrow_x}[\text{share}]$$

Fig. 8: Observability Predicate $P \downarrow_x$.

Remarkably, our proof of Theorem 2 leverages deep properties of Linear Logic, in particular the structure of the linear cut and co-contraction, allowing us to prove deadlock absence, even in a language with primitives exhibiting blocking behaviour, avoiding the use of extra mechanisms [47,33,48,10,25,76,31].

Strong Normalisation Establishing strong normalisation (SN) for concurrent process calculi is usually fairly challenging, particularly in the presence of name passing, recursion and higher-order shared state [32,16,83,49,69]. For example, with reference cells one may express general recursion with Landin's knot, and, in general, circular chains of references that may lead to divergence. However, our linear type system uses primitive recursion and corecursion, and excludes cyclic dependencies through state or session based interaction, allowing strong normalisation, and therefore livelock absence, to hold. Our proof relies on defining suitable linear logical relations, cf. [62,21,72], adapted to Classical Linear Logic [38,1,8], and crucially relying on a notion of reducibility up to interference that imposes stronger properties on the interpretation of the state modalities, and which allows the inductive proof of the Fundamental Lemma 2 to go through in the usual way. To this end, we extend our basic language with auxiliary constructs $\mathsf{cell}\ c(a.S)$ and $\mathsf{empty}\ c(a.S)$, which denote memory cells subject to interference from concurrent writers, allowed to take terms from the set $S \subseteq \{P \mid P \vdash_\eta a : \wedge A\}$. The intuition is that a take on the cell may always read any object from S, due to interference. We also consider the additional reduction (nondeterministic) rules (1)-(3), where in 1 and 2 we assume $P \in S$.

$$\begin{array}{lll} \mathsf{cell}\ c(a.S) \mid c \mid \mathsf{release}\ c & \to P \mid a \mid \mathsf{discard}\ a, & (1) \\ \mathsf{cell}\ c(a.S) \mid c \mid \mathsf{take}\ c(a'); Q & \to \mathsf{empty}\ c(a.S) \mid c \mid (P \mid a \mid \{a/a'\}Q) & (2) \\ \mathsf{empty}\ c(a.S) \mid c \mid \mathsf{put}\ c(a.P); Q & \to \mathsf{cell}\ c(a.S) \mid c \mid Q & (3) \end{array}$$

In this section, we thus consider reduction of $P \to Q$ to be the relation defined in Fig 7, extended with these rules. When a take or a release interacts with $\mathsf{cell}\ c(a.S)$, an arbitrary element P from the set S may be picked (rules (1) and (2)). In (3), a put $\mathsf{put}\ c(a.P); Q$ interacts with $\mathsf{empty}\ c(a.S)$ causing $\mathsf{empty}\ c(a.S)$ to evolve to $\mathsf{cell}\ c(a.S)$ (3). The following notion is also useful. A process P is *S-preserving on x* if $P \vdash_\eta x : \mathsf{U}_\bullet A$ or $P \vdash_\eta x : \mathsf{U}_\circ A$, and

- if $P \overset{*}{\to}\approx \mathsf{take}\ x(y); P'$ and $Q \in S$, then $Q \mid y \mid P'$ is S-preserving on x.
- if $P \overset{*}{\to}\approx \mathsf{put}\ x(y.P_1); P_2$, then $P_1 \in S$ and P_2 is S-preserving on x.

A set of processes T is S-preserving on x if and only for all $P \in T$, P is S-preserving on x. Intuitively a process P that uses a cell x is S-preserving on x if it only puts values from S on cell x. The notion of S-preservation, parametric on any S, brings explicit the conditions needed for safe interaction with a memory cell, subject to interference, while ensuring a state invariant S on the cell contents. We now introduce the logical predicate.

Definition 6 (Logical Predicate $[\![x : A]\!]_\sigma$). *By induction on the type A, we define the sets $[\![x : A]\!]_\sigma$ an shown in Fig. 9, such that $[\![x : \mathsf{U}_\bullet A]\!]_\sigma$ and $[\![x : \mathsf{U}_\circ A]\!]_\sigma$ are $[\![- : \wedge\overline{A}]\!]$-preserving on x.The definition is direct for the positive types A, for negative types B is given by orthogonality.*

The definition relies on Girard's notion of orthogonality $S^\perp \triangleq \{P \mid \forall Q \in S.\ P \ |x|\ Q \text{ is SN}\}$ [37]. Duality promotes succinctness in our definition: for negative types A, $[\![x : A]\!]_\sigma$ is defined as the orthogonal of the predicate for its dual $\overline{A}$ (positive) type. To handle polymorphic and inductive types, the logical predicate is indexed by a map σ that assigns reducibility candidates $R[x : A]$ to type variables. A reducibility candidate $R[x : A]$ is any set S of processes $P \vdash_\emptyset x : A$ such that P is SN and $S = S^{\perp\perp}$. We let $\mathcal{R}[- : A]$ be the set of all reducibility candidates $R[x : A]$ for some name x. The definition relies on a congruence relation $\approx$ extending $\leq$ with a complete set of commuting conversions, along standard lines [22,27,80]. It essentially plays the role of the labelled transition system in the proof of strong normalisation given in [62].

We extend the logical predicate to typing judgements $P \vdash_\eta \Delta; \Gamma$ by universal closure over the typing context and σ.

Definition 7 (Extended Logical Predicate $\mathcal{L}[\![\vdash_\eta \Delta; \Gamma]\!]_\sigma$). *We define $\mathcal{L}[\![\vdash_\eta \Delta; \Gamma]\!]_\sigma$ inductively on Δ, Γ and η as the set of processes $P \vdash_\eta \Delta; \Gamma$ s.t.*

$P \in \mathcal{L}[\![\vdash_\emptyset \emptyset; \emptyset]\!]_\sigma$ *iff* P *is SN.*
$P \in \mathcal{L}[\![\vdash_\emptyset \Delta, x : A; \Gamma]\!]_\sigma$ *iff* $\forall Q \in [\![x : \overline{A}]\!]_\sigma.\ Q\ |x : \overline{A}|\ P \in \mathcal{L}[\![\vdash_\emptyset \Delta; \Gamma]\!]_\sigma.$
$P \in \mathcal{L}[\![\vdash_\emptyset \Delta; \Gamma, x : A]\!]_\sigma$ *iff* $\forall Q \in [\![y : \overline{A}]\!]_\sigma.\ y.Q\ |!x : \overline{A}|\ P \in \mathcal{L}[\![\vdash_\emptyset \Delta; \Gamma]\!]_\sigma.$
$P \in \mathcal{L}[\![\vdash_{\eta, X(x,\boldsymbol{y}) \mapsto \Delta', x:Y; \Gamma} \Delta; \Gamma]\!]_\sigma$ *iff* $\forall Q \in \sigma(Y).\ \{Q/X\}P\ \in \mathcal{L}[\![\vdash_\eta \Delta; \Gamma]\!]_\sigma.$

We now state the Fundamental Lemma (2) from which Theorem 3 follows.

Lemma 2 (Fundamental Lemma). *If $P \vdash_\eta \Delta; \Gamma$, then $P \in \mathcal{L}[\![\vdash_\eta \Delta; \Gamma]\!]_\sigma$.*

Proof. (Sketch) By induction on $P \vdash_\eta \Delta; \Gamma$. For cases [Tcell] and [Tempty], we show that $\mathsf{cell}\ c(a.S)$ and $\mathsf{empty}\ c(a.S)$ respectively simulate $\mathsf{cell}\ c(a.P)$ (where $P \in S$) and $\mathsf{empty}\ c$, when composed with any S-preserving on c usages. To handle one of the most challenging cases, [Tsh] we prove, for all S, and all S-preserving on x processes P_1 and P_2, that $\mathsf{cell}\ c(a.S)\ |c|\ \mathsf{share}\ c\ \{P_1 \ ||\ P_2\}$ (1) is simulated by $(\mathsf{cell}\ c(a.S)\ |c|\ P_1)\ ||\ (\mathsf{cell}\ c(a.S)\ |c|\ P_2)$ (2). This allows us to infer that if (2) is SN, then so it is (1). When $S = [\![a : \wedge\overline{A}]\!]_\sigma$, the i.h. yields $(\mathsf{cell}\ c(a.S)\ |c|\ P_i)$ SN, hence we conclude (2) SN. Similarly for [TshL], [TshR].

Theorem 3 (Strong Normalisation). *If $P \vdash_\emptyset \emptyset; \emptyset$, then P is SN.*

$$
\begin{array}{ll}
[\![x : X]\!]_\sigma & \triangleq \sigma(X)[x] \\
[\![x : \mathbf{1}]\!]_\sigma & \triangleq \{P \mid P \approx \mathsf{close}\ x \text{ and } P \text{ is SN}\}^{\perp\perp} \\
[\![x : A \otimes B]\!]_\sigma & \triangleq \{P \mid \exists P_1, P_2.\ P \approx \mathsf{send}\ x(y.P_1); P_2 \text{ and} \\
& \qquad P_1 \in [\![y : A]\!]_\sigma \text{ and } P_2 \in [\![x : B]\!]_\sigma\}^{\perp\perp} \\
[\![x : A \oplus B]\!]_\sigma & \triangleq \{P \mid \exists Q.\ P \approx x.\mathsf{inl}; Q \text{ and } \ Q \in [\![x : A]\!]_\sigma \ \text{ or} \\
& \qquad\qquad P \approx x.\mathsf{inr}; Q \text{ and } Q \in [\![x : B]\!]_\sigma\}^{\perp\perp} \\
[\![x : !A]\!]_\sigma & \triangleq \{P \mid \exists Q.\ P \approx !x(y); Q \text{ and } Q \in [\![y : A]\!]_\sigma\}^{\perp\perp} \\
[\![x : \exists X.A]\!]_\sigma & \triangleq \{P \mid \exists Q, S\ \in \mathcal{R}[- : B].\ P \approx \mathsf{sendty}\ x(B); Q \text{ and} \\
& \qquad\qquad\qquad Q \in [\![x : A]\!]_{\sigma[X \mapsto S]}\}^{\perp\perp} \\
[\![x : \mu X.\ A]\!]_\sigma & \triangleq (\bigcap\{S \in \mathcal{R}[- : \mu X.A] \mid \mathsf{unfold}_\mu\ x; [\![x : A]\!]_{\sigma[X \mapsto S]} \subseteq S\})^{\perp\perp} \\
[\![x : \wedge A]\!]_\sigma & \triangleq \{P \mid \exists Q.\ P \approx \mathsf{affine}\ x; Q \text{ and } Q \in [\![x : A]\!]_\sigma\}^{\perp\perp} \\
[\![x : \mathsf{S}_\bullet A]\!]_\sigma & \triangleq \{P \mid P \approx \mathsf{cell}\ x(y.[\![y : \wedge A]\!]_\sigma) \text{ and } P \text{ is SN}\}^{\perp\perp} \\
[\![x : \mathsf{S}_\circ A]\!]_\sigma & \triangleq \{P \mid P \approx \mathsf{empty}\ x(y.[\![y : \wedge A]\!]_\sigma) \text{ and } P \text{ is SN}\}^{\perp\perp} \\
[\![x : B]\!]_\sigma & \triangleq [\![x : \overline{B}]\!]_\sigma^\perp \ \text{ (B negative type)}
\end{array}
$$

Fig. 9: Logical Predicate $[\![x : A]\!]_\sigma$.

4 Typeful Concurrent Programming in CLASS

In this section, we discuss the expressiveness of CLASS's type system, going through a sequence of illustrative realistic concurrent programming idioms.

Sharing a Linear Session. Our first example illustrates how objects subject to a linear usage protocol and satisfying an invariant may be shared among multiple concurrent clients by serialising linear usages using a mutex cell, alternating ownership from the cell to clients and back at the invariant state, a commonly used discipline to implement and reason about resource sharing (see, e.g., [39,17,9]). We illustrate with a basic toggle switch with two states - On and Off - the resource invariant is the state Off, and two operations #turnOn and #turnOff that must be executed in strict linear sequence (Fig. 10). The toggle protocol, defined by type Off, offers the single option #turnOn, after which it evolves to On. Conversely, type On offers the single option #turnOff, after which it evolves to an affine Off. The toggle process at t is defined by two mutually corecursive processes on(t) and off(t), which define the expected behaviour, and comply with types On and Off.

Process main() introduces a mutex cell c storing an affine toggle object at the invariant type $\wedge$Off. It then shares it with two concurrent clients, each acquires the toggle in the invariant type and uses the linear protocol independently. After their linear interaction, they put back the toggle, the type system ensures that this can only happen when the invariant (given by the cell type) holds. When they are done, both clients release their respective usages of c, which ultimately leads to the cell being deallocated and the (affine) toggle to be discarded.

```
type corec Off = &{|#turnOn : On}
type corec On  = &{|#turnOff : ∧Off}

off(t) ⊢ t : Off
off(t) = case t {|#turnOn : on(t)}

on(t) ⊢ t : On
on(t) = case t {|#turnOff :
                    affine t; off(t)}

client1(c) ⊢ c : S̅•̅O̅f̅f̅
client1(c) = take c(t);
             #turnOn t; #turnOff t;
             put c(t); release c
```

```
client2(c) ⊢ c : S̅•̅O̅f̅f̅
client2(c) = take c(t);
             #turnOn t; #turnOff t;
             #turnOn t; #turnOff t;
             put c(t); release c

main()   ⊢ ∅
main()   = cut {cell c(t.affine t; off(t))
             |c|
             share c {
                client1(c)   ||
                client2(c) }}
```

Fig. 10: Sharing a Linear Toggle Switch

```
type rec SList(A) = S•List(A)
type rec List(A)  = ⊕{
                      |#Null : 1,
                      |#Next : ∧A ⊗ SList(A)}

nil(l) ⊢ l : ∧List(A)
nil(l) = affine l; #Null l;  close l

cnext(a, c, l) ⊢ a: ∨ A̅, c:S̅L̅i̅s̅t̅(̅A̅)̅, l: ∧ List(A)
cnext(a, c, l) = affine l;
                 #Next l;
                 send l(a);
                 fwd l c
```

```
append(c, l′, c′) =
 take c(l);
 case l {
   |#Null :
     wait l;  put c(l′);  fwd c c′
   |#Next :
     recv l(a);
     cut {
       append(l, l′, x)
       |x|
       put c(y.cnext(a, x, y));
       fwd c c′ }}
```

Fig. 11: A Linked List with an Append In-Place Operation.

We have also developed CLASS code for a generic (polymorphic) wrapper factory that, for any affine corecursive protocol, generates a wrapper to a general invariant-based sharing interface.

Linked Lists, Update In-Place. In this example, we show how inductive/-coinductive types combine harmoniously with CLASS state modalities to type linked data structures with memory-efficient updates in-place. Specifically, we show how to code a linked list, parametric on the type A of its affine values, with update in-place append (Fig. 11). An object of type SList(A) is a (full) cell storing a List(A) object. An object of type List(A) is a session that either selects #Null (the list is empty), in which case it closes; or selects #Next, in which case it sends an affine session $\wedge A$ representing the head element and continues as the tail SList(A). Process nil(l) - defines an empty list at l - and process cnext(a, c, l) - constructs a nonempty list l with head a and tail c. For example, a list with elements a, b stored at $c_1 : \mathsf{S}_\bullet\mathsf{List}(A)$ is represented

$$\mathsf{cut}\{\ \mathsf{cell}\ c_1(l_1.\mathsf{cnext}(a, c_2, l_1))\ |c_2|\ \mathsf{cell}\ c_2(l_2.\mathsf{cnext}(b, c_s, l_2))\ |c_s|\ \mathsf{cell}\ c_s(l_0.\mathsf{nil}(l_0))\}$$

Process $\mathsf{append}(c, l', c') \vdash c : \overline{\mathsf{SList}(A)}, l' : \overline{\mathsf{List}(A)}, c' : \mathsf{SList}(A)$ produces on c' the result of appending l (in place) to c. It takes the list l stored in c, and then performs case analysis on l. If l selects $\#\mathsf{Null}$, it simply replaces the previous null node of c by l' and forwards the updated cell c to the output c'. This corresponds to the recursion base case in which the list l is empty.

If l selects $\#\mathsf{Next}$, in which case l has at least one element, one receives at l the node element $a : \vee\overline{A}$, and corecursively call append l' to the tail $l : \overline{\mathsf{SList}(A)}$ and puts back in c element a and tail x "returned" by the call. Notice that x is exactly x (by forwarding), which was passed along linearly. Remarkably, the $\mathsf{append}(c, l', c')$ operation just defined may be safely applied concurrently to the same shared linked list, with the final result being the correct one (some serialisation of the appends), without deadlocks or livelocks. It is also interesting to see how the type system forbids a list to be appended to itself.

We have also developed many other in-place operations on linked data structures, such as insertion sort, and other kinds of linked structures such as queues and binary search trees. In the next examples we discuss a shared queue ADT with a fine-grained locking discipline and O(1) enqueue and dequeue operations.

A Concurrent Shareable Buffered Channel. We illustrate increased degrees of sharing in a mutable data structure with various references pointing to different parts of it, how the CLASS type system may express interfaces that talk about different client views for using a stateful object, and the use of polymorphism to implement information hiding ensuring that client code will never break the representation invariants of stateful ADTs, particularly challenging when aliasing and sharing are involved.

More concretely, we consider a shareable buffered channel (Fig. 12), and provide a realistic and efficient implementation [56] based on a message queue represented by a linked list with update-in-place (cf. Section 4 above) and two independent pointers: one to the head of the list, used for receiving, and another to the tail, used for sending. The operations are executed in O(1) time. Moreover we provide a typing with two separate send and receive views, which may be used by an arbitrary number of concurrent clients. In particular, when the list is nonempty, both send and receive run in true concurrency (asynchronously), without blocking each other, thanks to fine-grained locking.

The buffered channel type $\mathsf{BChan}(M)$, where M is the type of messages, offers two views: $\mathsf{SendT}(M)$ and $\mathsf{RecvT}(M)$, interfaces for sender and receiver endpoint clients. These views are exposed with a par ($\mathbin{⅋}$), since they share an underlying resourceful structure. In fact, they could not be exported using a tensor ($\otimes$); it is interesting to notice how the type system imposes these constraints, important to ensure deadlock freedom. The representation type of both views is $\mathit{Rep} = \mathsf{S}_\bullet\mathsf{SList}(M)$ (see Section 4), hidden behind the SV and RV existential types [29,58]; sending clients use a cell storing a reference to the tail node of

```
type BChan(M) = SendT(M) ⅋ RecvT(M)
type SendT(M) = ∃SV.!MenuS(M, SV) ⊗ SV
type RecvT(M) = ∃RV.!MenuR(M, RV) ⊗ RV

type MenuS(M, SV) = & {
   |#Send : SV ⊸ ∧M ⊸ SV,
   |#Share : SV ⊸ (SV ⅋ SV),
   |#Free : SV ⊸ 1 },

type MenuR(M, RV) = & {
   |#Recv : RV ⊸ (Maybe(∧M) ⊗ RV),
   |#Share : RV ⊸ (RV ⅋ RV),
   |#Free : RV ⊸ 1 }

Rep = SV = RV = S•SList(M)
```

```
msend(me) =
recv me(tailptr);
recv me(a);
take tailptr(c);
take c(l);
cut {
   cell c′(l)
   |c′|
   share c′ {
      put c(l′.cnext(a, c′, l′));
      release c′
      ||
      put tailptr(c′);
      send me(tailptr);
      close me}}
```

Fig. 12: A Concurrent Shareable Buffered Channel.

the queue; receiving clients use a cell storing a reference to the head node of the queue.

Clients use the buffer through references of abstract type SV and RV and replicated menus !MenuS(M, SV) and !MenuR(M, RV). Both menus export the options #Share and #Free to allow sharing and release of the views. To send, a client selects #Send, sends his handle (of opaque type SV), the message to send and receives the (linear) handle back. In this implementation, receive is non-blocking, so operation #Recv returns a Maybe$(\wedge M)$ value: the client receives either #Nothing (if the buffer is empty) or #Just followed by a message a, otherwise. In 4 we discuss the implementation, in CLASS, of (Hoare style) monitors with conditions, which would allow a blocking receive to be implemented.

Process msend(me) implements the #Send "method". It first receives the sending view handle (of concrete type *Rep*), which is a cell with the *tailptr*, and the message a to be sent. Then, a new cell c' with nil (l) is created, the current tail of the list c is updated with a new node storing a and pointing to c'. Finally, the *tailptr* cell is updated to point to the new tail node c' of the linked list.

Dining Philosophers. A resource hierarchy solution for the dining philosophers problem [34] requires forks to be acquired in a defined order. We "encode" such order in CLASS with an explicit (necessarily) acyclic structure, which informs the type system about the code safety. This allows us to define a correct implementation that satisfies deadlock freedom by pure linear logic typing. More concretely, we organise the forks in a linked chain defined by the inductive types rec Fork $=$ S$_\bullet$Node and rec Node $= \oplus\{$#Null : $\mathbf{1}$, #Next : Fork$\}$.

Any fork in the chain may be shared by an arbitrary number of philosophers, cocontraction ensures that philosophers cannot communicate between themselves via any other channel, all synchronisation must happen via the chained

```
putNull(f, f′) ⊢ f : U∘‾Node‾, f′ : Fork
putNull(f, f′) ≜ put f(n.null(n)); fwd f f′

eat(f, f′) ⊢ f : ‾Fork‾, f′ : Fork
eat(f, f′) ≜
   take f(n);
   case n {
      |#Null :
      wait n; putNull(f, f′)
      |#Next :
      take n(m);
      put n(m); put f(n′.next(n, n′));
      fwd f f′}

eat2(f, f′) ⊢ f : ‾Fork‾, f′ : Fork
eat2(f, f′) ≜
   take f(n);
   case n {
      |#Null :
      wait n; putNull(f, f′)
      |#Next :
      cut {
         takeLast(n, x)
         |x|
         recv x(m); wait x;
         put f(n′.next(m, n′));
         fwd f f′}
```

Fig. 13: The Dining Philosophers.

forks. Furthermore, the chain can be resized and grow unboundedly to accommodate an arbitrary number of philosophers. If a philosopher successfully takes a fork f_i, he can then take any fork f_j, with $i < j$; crucially, he must follow the path dictated by the chain, hence cannot acquire forks f_j with $j < i$. In Fig. 13 we define the eat operation, which allows each philosopher P_i, with $0 \leq i < k-1$ to eat: it acquires two consecutive forks in the chain. And eat2, which is the specific eating operation for the symmetry breaker P_{k-1}: it acquires the first fork, and traverses the chain to acquire the last with $\mathsf{takeLast}(n, x) \vdash n : \overline{\mathsf{Fork}}, x : \mathsf{Fork} \otimes \mathbf{1}$.

A Barrier for N threads. We describe in Fig. 14 a CLASS implementation of a simple barrier, parametric on the number N of threads to synchronise. We find it interesting to model the "real" code shown in the Rust reference page for std::sync::Mutex [46]. The code uses if-then-else and primitive integers, as offered in our implementation, that could be defined as idioms in CLASS. We represent a barrier by a mutex cell storing a pair consisting of an integer n, holding the number of threads that have not yet reached the barrier, and a stack s of waiting threads, each represented by a session of *affine* type $\wedge\bot$ (so they will be safely aborted if at least one thread fails to reach the barrier).

The type Barrier of the barrier is $\mathsf{S}_\bullet\mathsf{BState}$, where $\mathsf{BState} \triangleq \mathsf{Int} \otimes \wedge\mathsf{List}(\wedge\bot)$. Initially the barrier is initialised with $n = N$ threads and an empty stack, so that the invariant $n + depth(s) = N$ holds during execution. Each $\mathsf{thread}(c; i)$ acquires the barrier c and checks if it is the last thread to reach the barrier (if $n == 1$): in this case, it awakes all the waiting threads ($\mathsf{awakeAll}(w_s)$) and resets the barrier. Otherwise, it updates the barrier by decrementing n and pushing its continuation into the stack (the continuation for thread i just prints "finished"). The following process $\mathsf{main}() \vdash \emptyset$ creates a new barrier c and spawns N threads, each labelled

```
init($w_s$) ⊢ $w_s$ : ∧BState
init($w_s$) ≜
   affine $w_s$; send $w_s(N)$; affine $w_s$; nil($w_s$)

awakeAll($w_s$ : $\overline{\text{List}(\wedge\bot)}$)
awakeAll($w_s$) ≜
   case $w_s$ {
      #Nil : wait $w_s$; 0
      #Cons :
      recv $w_s(w)$;
      par {close $w$ || awakeAll($w_s$)}

spawnAll($c; i, n$) ⊢ $c$ : $\overline{\text{Barrier}}$; $i$ : $\overline{\text{Int}}$, $n$ : $\overline{\text{Int}}$
spawnAll($c; i, n$) ≜
   if ($n == 0$) { release $c$}
   { share $c$ {
         thread($c; i$)
         ||
         spawnall($c; i+1, n-1$)}}
```

```
thread($c; i$) ⊢ $c$ : $\overline{\text{Barrier}}$; $i$ : $\overline{\text{Int}}$
thread($c; i$) =
   println $i$ + ": waiting.";
   take $c(w_s)$; recv $w_s(n)$;
   if ($n == 1$) {
      par {
         println $i$ + ": finished.";
         awakeAll($w_s$)
         ||
         put $c(w'_s$.init($w'_s$));
         release $c$}}
   { cut {
         affine $w$; wait $w$;
         println $i$ + ": finished."; 0
         |$w$| put $c(w'_s$.affine $w'_s$;
                     send $w'_s(n-1)$;
                     affine $w'_s$;
                     cons($w, w_s, w'_s$));
         release $c$}}
```

Fig. 14: A Barrier for N Threads

by a unique id i: main() ≜ cut { cell $c(w_s$.init$(w_s))$ |c| spawnAll($c; 0, N$) }. Again, our type system statically ensures that the code does not deadlock or livelock.

A Hoare Style Monitor. A Hoare style monitor is a well-know powerful programming abstraction [39], allowing concurrent operations on shared data to be coordinated in a sound way, so that it always satisfy a correctness invariant. The key essential idea is that concurrent client threads use the monitor lock to access the protected state in mutual exclusion, but may also wait (via a *await* primitive) inside the monitor until the state satisfies specific (pre-)conditions, while transferring state ownership to other threads potentially responsible for establishing such conditions and announcing it (via a *notify* primitive).

We discuss a CLASS implementation of a monitor, sketching the main components and how they are typed (Fig. 15). We consider a counter with value n, with increment #Inc and decrement #Dec operations, and subject to the invariant $n \geq 0$. The type of the counter CounterI exposes two separate, coinductively defined, client interfaces DecI and IncI for decrementing and incrementing.

While the #Inc operation is synchronous, the #Dec operation is always called asynchronously by passing a continuation (of type ContDec). This allows decrementers to wait inside the monitor for condition NZ ($n > 0$) when $n = 0$. The condition NZ is represented by a wait queue of type WaitQ. The representation type of the monitor (Rep) holds the counter value and the wait queue. Each node in the wait queue stores information, of type ContDecW, for the waiting thread.

```
type corec IncI ≜ &{|#Inc : IncI, |#End : ⊥}
type corec DecI ≜
   ∨ & {|#Dec : ∨(ContDec ⊸ ⊥), #End : ⊥}
type corec ContDec ≜ ∨(DecI ⊗ 1)
type CounterI ≜ DecI ⅋ IncI

type rec Rep ≜ (!Int) ⊗ WaitQ
type rec WaitQ ≜ ∧ ⊕ {|#Null : 1, |#Next : NodeQ}
type rec NodeQ ≜ S•(ContDecW ⊗ WaitQ)
type rec ContDecW ≜ ∧(∧Rep ⊸ ∧Rep ⊗ DecI ⊸ ⊥)
```

$$\begin{array}{l}
\mathsf{awaitNZ} \vdash m : \mathsf{U}_\circ\overline{\mathsf{Rep}},\\
\qquad\qquad n : \overline{\mathsf{!Int}}, w : \overline{\mathsf{WaitQ}}, cc : \overline{\mathsf{ContDecW}}\\
\mathsf{notifyNZ} \vdash m : \mathsf{U}_\circ\overline{\mathsf{Rep}}, s : \overline{\mathsf{Rep}}, m' : \mathsf{S}_\bullet\mathsf{Rep}\\
\mathsf{incloop} \vdash iv : \overline{\mathsf{IncI}}, m : \mathsf{U}_\bullet\overline{\mathsf{Rep}}
\end{array}$$

```
awaitNZ(m, n, w, cc) ≜
   put m(w′.affine v;
      send w′(n);
      consWQ(cc, w, w′));
      release m

incloop(iv, m) ≜
   case iv {
   #Inc : take m(r);
          recv r(n);
   cut {
   send s(n + 1); fwd s r
   |s| notifyNZ(m, s, m′)
   |m′| incloop(iv, m′) }
   #End : wait iv;
          release m}
```

Fig. 15: Implementing a Counter Monitor with Await / Notify.

Every such ContDecW objects stores (1) the pending action on the internal monitor state (of type ∧Rep ⊸ ∧Rep), to be executed after await returns, and (2) a callback to the continuation provided by the external client in the asynchronous call (of type DecI ⊸ ⊥).

The awaitNZ(m, n, w, cc) process implements the monitor wait operation, used in the #Dec operation. It receives the (empty) cell usage m to the monitor state, the integer value n (where $n = 0$), a reference w to the wait queue, and the continuation cc, it pushes a new node in the queue and puts the monitor state back, unlocking the cell m, and releases m. The incloop(iv, m) process implements the counter IncI interface. The call to notifyNZ(m, s, m') after incrementing n will cause a waiting DecI thread to be awaken (if any), and continue by applying the pending action to the Rep state s in which $n > 0$ holds, before passing the updated state m' to the incloop recursive call. Affinity plays a key role, allowing all data structures, including waiting continuations to be safely discarded, at the end of any computation. We have only shown here some code snippets, the complete code is available in the CLASS distribution.

Our examples illustrate how our system types non-trivial concurrent code, akin to real system-level code, involving higher-order state, rich sharing and ownership transfer patterns, while ensuring deadlock, livelock freedom and memory safety. Our typing of sharing imposes that only a single bundle of linear resources may be shared by two independent threads. As our examples show, code can often be structured in that way, so that bundles of many linear resources may be safely shared by monitor-like structures, exposing informative typed interfaces.

The feasibility of CLASS is corroborated by our implementation [68] of a fully-fledged type checker and interpreter, developed in Java (∼15k), and packaged with an extensive CLASS library of code and test suites (∼10k), including all the examples in this paper. Type checking is decidable in polynomial time, using a minimal type annotation, only on cut-bound names and function parameters, the multiplicative rules are handled by lazy context splitting (cf. [41]). The type checker ensures that corecursive calls are done on a session *hereditarily descendent* from the corecursion parameter, a condition motivated by our SN result (Theorem 3). But we also support an unsafe corecursion mode, in which this check is turned off, to type programs defined by general corecursion.

The type checker supports useful type inference and reconstruction abilities. The interpreter uses java.util.concurrent.* package [53], using primitives such as fine-grained locks and condition variables to emulate the synchronous interactions of CLASS sessions and a cached thread pool to manage the life cycle of short-lived threads. Cell deallocation is implemented by reference counting, incremented on each share and decremented on each release. Forwarding redirects the clients of a shared cell through a chain of forwarding pointers (cf. [9]).

5 Related Work

Many resource-aware logics and type systems to tame shared state and interference have been proposed [3,18,57,77,44,17,60,61,24]. These systems adopt some form of linearity and/or affinity to resourceful programming [75,30] and to model failures/exceptions [28,59,20,36,52]. In CLASS, linearity allows us to control state sharing, whereas affinity is useful to ensure memory safety and to represent safely finalizable or abortable computations. The hereditary session-discarding behaviour of affine sessions, modelled by rule [$\wedge\vee$d], is also present in other works, e.g. [6,59,20].

CLASS builds on top of the PaT correspondence with Linear Logic [22,27,80], the logical principles for the state modalities being inspired by DiLL [35]. Recent works [43,9,10,7,50,64,67] also address the problem of sharing and nondeterminism in the setting of session-based PaT. In [67], reference cells may only store replicated sessions (of type $!A$), thus cannot refer to linear entities such as other cells or linear sessions, hence cannot represent many realistic programming idioms that CLASS does (see Section 4). Accommodating linear state in a pure PaT approach is thus addressed in this work with a novel, more fundamental approach. Furthermore, in [67], recursion is obtained via a system-F style encoding [79], which cannot model inductive stateful structures with updates in-place as we do with CLASS native inductive/coinductive types.

The take/put operations of CLASS relate with Concurrent Haskell MVars [45] and the acquire/release operations of the manifest sharing session-typed language SILL$_S$ [9,10]. Sharing in SILL$_S$ is based on shift modalities to move from shared to linear mode and back, and contraction principles to alias shared sessions. In CLASS we explore DiLL modalities and cocontraction principles [35] to express sharing of linear state and put / take protocols of mutex memory cells of invariant type. The work [10] ensures deadlock-freedom by relying on programmer provided partial orders on events [55,33,26], whereas in CLASS deadlock-freedom follows the same simple and general inductive argument of the corresponding result in e.g. [22], thanks to the logical character of the new proof rules (DiLL cocontraction, that enjoys cut-elimination). The work [64] introduces the language CSLL, by extending linear logic with coexponentials that support a notion of shared state, with a quite different approach than ours. CSLL does not claim the ability to naturally express shared linked data structures with update in-place and fine-grained locking, as CLASS does. Nevertheless, it is natural to define in CLASS sessions exporting weakening, sharing and dereliction capabilities for linear behaviours, as in our shared buffer example.

Recently, the work [43] develops λ_{lock}, a substructural-typed λ-calculus with higher-order locks, which enjoys deadlock-freedom by imposing a set of high-level principles that guarantee acyclicity of the lock-sharing topologies, and which follow in CLASS as a consequence of its logical-motivated type system and DiLL's cocontraction. This work also extends λ_{locks} with partial orders in which a resource can shared by more than two concurrent threads. None of the models in [43,9,10,64] addresses livelock absence or memory safety, as CLASS does.

As far as we are aware, CLASS is a first proposal integrating shared state and recursion in a language based on PaT and Linear Logic, while guaranteeing strong normalisation. Least/greatest fixed points in Linear Logic were studied in [8], which inspired the development of recursion in [54,73], our treatment of recursion draws inspiration on [73]. Several works exploit the technique of logical relations to establish strong normalisation for concurrent process calculi [1,83,69,16,62]. The work [16] proves strong normalisation for a language with higher-order store with a type and effect system that stratifies memory into regions so as to preclude circularities. Interestingly, in CLASS such stratification is implicitly guaranteed by the acyclicity inherent to Linear Logic. Linear logical relations were studied in [62,21,72,74]. In this work we recast and extend the technique to Classical Linear Logic, exploring orthogonality [38,8,1], and demonstrate, using a specially devised technique of interference-sensitive reducibility, how logical relations scale to accommodate shared state.

6 Concluding Remarks

We have introduced CLASS, a session-based language founded on a propositions-as-types interpretation of Second-Order Classical Linear Logic, extended with recursion, affine types, first-class mutex cells and shared linear state. We believe that CLASS is the first proposal of a language of its kind to provide the following three strong properties by static typing: well-typed CLASS programs enjoy progress, hence never deadlock, do not leak memory and always terminate.

CLASS metatheoretical properties are obtained in a compositional and modular way, by leveraging the key features of propositions-as-types, from which the operational semantics and type system also emerges. In CLASS, types and process have a consistent proof-theoretical behaviour: typed program constructs correspond exactly to proof rules, with a proper compositional semantics via logical relations (Section 3). Programs are composed by plugging basic constructs with the cut rule, and all interaction principles are captured by principal cut reductions that act locally in proofs/type derivations (Def. 4). We also obtain an algebraic system based on proof simplification to reason about program (observational) equivalence, due to confluence (cf. [65]).

Besides the foundational relevance of our work, we also argued how CLASS can cleanly express realistic concurrent higher-order programming idioms, with many compelling examples. Any type system introduces conservative restrictions on its language, but we believe that CLASS offers an interesting balance between the strong properties it ensures by typing and its expressiveness. In fact, we find CLASS type system helpful to guide the development of safe concurrent idioms, with a fairly light type annotation burden. As future work, we would like to investigate several possible refinements of the CLASS type discipline, namely, allowing finer-grained resource-access policies to be expressed, and exploring the integration of dependent and refinement types [71,51], enhancing the logical expressiveness of the basic type system.

References

1. Abramsky, S.: Computational Interpretations of Linear Logic. Theoret. Comput. Sci. **111**(1–2), 3–57 (1993)
2. Abramsky, S., Gay, S.J., Nagarajan, R.: Interaction categories and the foundations of typed concurrent programming. In: NATO ASI DPD. pp. 35–113 (1996)
3. Ahmed, A., Fluet, M., Morrisett, G.: L^3: A linear language with locations. Fundam. Inf. **77**(4), 397–449 (Dec 2007)
4. Andreoli, J.M.: Logic programming with focusing proofs in linear logic. J. Logic Comput. **2**(3), 197–347 (1992)
5. Andreoli, J.M.: Logic Programming with Focusing Proofs in Linear Logic. J. Log. Comput. **2**(3), 297–347 (1992)
6. Asperti, A., Roversi, L.: Intuitionistic light affine logic. ACM Transactions on Computational Logic (TOCL) **3**(1), 137–175 (2002)
7. Atkey, R., Lindley, S., Morris, J.G.: Conflation confers concurrency. In: A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, pp. 32–55. Springer (2016)
8. Baelde, D.: Least and greatest fixed points in linear logic. TOCL **13**(1) (Jan 2012)
9. Balzer, S., Pfenning, F.: Manifest sharing with session types. Proc. ACM Program. Lang. **1**(ICFP) (Aug 2017)
10. Balzer, S., Toninho, B., Pfenning, F.: Manifest deadlock-freedom for shared session types. In: Caires, L. (ed.) Programming Languages and Systems. pp. 611–639. Springer International Publishing, Cham (2019)
11. Barber, A.: Dual Intuitionistic Linear Logic. Tech. Rep. LFCS-96-347, Univ. of Edinburgh (1996)
12. Beffara, E.: A Concurrent Model for Linear Logic. ENTCS **155**, 147–168 (2006)
13. Beffara, E.: An algebraic process calculus. In: Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science. p. 130–141. LICS '08, IEEE Computer Society, USA (2008)
14. Bellin, G., Scott, P.: On the π-calculus and linear logic. Theoret. Comput. Sci. **135**(1), 11–65 (1994)
15. Benton, P.N.: A mixed linear and non-linear logic: Proofs, terms and models. In: International Workshop on Computer Science Logic. pp. 121–135. Springer (1994)

16. Boudol, G.: Typing termination in a higher-order concurrent imperative language. Information and Computation **208**(6), 716–736 (2010)
17. Brookes, S., O'Hearn, P.W.: Concurrent Separation Logic. ACM SIGLOG News **3**(3), 47–65 (2016)
18. Caires, L.: Logical Semantics of Types for Concurrency. In: International Conference on Algebra and Coalgebra in Computer Science. pp. 16–35. CALCO'07, Springer LNCS 4624 (2007)
19. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Relational parametricity for polymorphic session types. Tech. Rep. CMU-CS-12-108, Carnegie Mellon Univ. (2012)
20. Caires, L., Pérez, J.A.: Linearity, control effects, and behavioral types. In: Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201. p. 229–259. Springer-Verlag, Berlin, Heidelberg (2017)
21. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Behavioral polymorphism and parametricity in session-based communication. In: Proceedings of the 22nd European Conference on Programming Languages and Systems. p. 330–349. ESOP'13, Springer-Verlag, Berlin, Heidelberg (2013)
22. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory. pp. 222–236. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
23. Caires, L., Pfenning, F., Toninho, B.: Towards concurrent type theory. In: Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation. p. 1–12. TLDI '12, Association for Computing Machinery, New York, NY, USA (2012)
24. Caires, L., Seco, J.a.C.: The type discipline of behavioral separation. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 275–286. POPL '13, Association for Computing Machinery, New York, NY, USA (2013)
25. Caires, L., Vieira, H.T.: Conversation types. In: Castagna, G. (ed.) Programming Languages and Systems. pp. 285–300. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
26. Caires, L., Vieira, H.T.: Conversation types. Theor. Comput. Sci. **411**(51-52), 4399–4440 (2010)
27. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science **26**(3), 367–423 (2016)
28. Carbone, M., Honda, K., Yoshida, N.: Structured Interactional Exceptions in Session Types. In: CONCUR 2008. LNCS, vol. 5201, pp. 402–417. Springer (2008)
29. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. ACM Computing Surveys (CSUR) **17**(4), 471–523 (1985)
30. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 48–64. OOPSLA '98, Association for Computing Machinery, New York, NY, USA (1998)
31. Dardha, O., Gay, S.J.: A new linear logic for deadlock-free session-typed processes. In: Baier, C., Dal Lago, U. (eds.) Foundations of Software Science and Computation Structures. pp. 91–109. Springer International Publishing, Cham (2018)
32. Demangeon, R., Hirschkoff, D., Sangiorgi, D.: Mobile processes and termination. In: Semantics and Algebraic Specification, pp. 250–273. Springer (2009)
33. Dezani-Ciancaglini, M., de'Liguoro, U., Yoshida, N.: On progress for structured communications. In: Barthe, G., Fournet, C. (eds.) Trustworthy Global Computing. pp. 257–275. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

34. Dijkstra, E.W.: Hierarchical ordering of sequential processes. In: The origin of concurrent programming, pp. 198–227. Springer (1971)
35. Ehrhard, T.: An introduction to differential linear logic: proof-nets, models and antiderivatives. Mathematical Structures in Computer Science **28**(7), 995–1060 (2018)
36. Fowler, S., Lindley, S., Morris, J.G., Decova, S.: Exceptional asynchronous session types: session types without tiers. Proceedings of the ACM on Programming Languages **3**(POPL), 1–29 (2019)
37. Girard, J.Y.: Linear logic. Theoret. Comput. Sci. **50**(1), 1–102 (1987)
38. Girard, J.Y.: Linear logic. Theoretical computer science **50**(1), 1–101 (1987)
39. Hoare, C.A.R.: Monitors: An operating system structuring concept. Commun. ACM **17**(10), 549–557 (1974)
40. Hoare, C.A.R.: Towards a theory of parallel programming. In: The origin of concurrent programming, pp. 231–244. Springer (1972)
41. Hodas, J.S., Miller, D.: Logic programming in a fragment of intuitionistic linear logic. Information and computation **110**(2), 327–365 (1994)
42. Howard, W.A.: The formulae-as-types notion of construction. In: Seldin, J.P., Hindley, J.R. (eds.) To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 479–490. Academic Press (1980)
43. Jacobs, J., Balzer, S.: Higher-order leak and deadlock free locks. Proceedings of the ACM on Programming Languages **7**(POPL), 1027–1057 (2023)
44. Jacobs, J., Balzer, S., Krebbers, R.: Connectivity graphs: a method for proving deadlock freedom based on separation logic. Proc. ACM Program. Lang. **6**(POPL), 1–33 (2022)
45. Jones, S.P., Gordon, A., Finne, S.: Concurrent Haskell. In: POPL. vol. 96, pp. 295–308. Citeseer (1996)
46. Klabnik, S., Nichols, C.: The Rust Programming Language (2021)
47. Kobayashi, N.: A type system for lock-free processes. Information and Computation **177**(2), 122–159 (2002)
48. Kobayashi, N.: A new type system for deadlock-free processes. In: International Conference on Concurrency Theory. pp. 233–247. Springer (2006)
49. Kobayashi, N., Sangiorgi, D.: A hybrid type system for lock-freedom of mobile processes. ACM Transactions on Programming Languages and Systems (TOPLAS) **32**(5), 1–49 (2008)
50. Kokke, W., Morris, J.G., Wadler, P.: Towards races in linear logic. In: Riis Nielson, H., Tuosto, E. (eds.) Coordination Models and Languages. pp. 37–53. Springer International Publishing, Cham (2019)
51. Krishnaswami, N.R., Pradic, P., Benton, N.: Integrating linear and dependent types. ACM SIGPLAN Notices **50**(1), 17–30 (2015)
52. Lagaillardie, N., Neykova, R., Yoshida, N.: Stay safe under panic: Affine rust programming with multiparty session types. arXiv preprint arXiv:2204.13464 (2022)
53. Lea, D.: Concurrent programming in Java: design principles and patterns. Addison-Wesley Professional (2000)
54. Lindley, S., Morris, J.G.: Talking bananas: structural recursion for session types. In: Garrigue, J., Keller, G., Sumii, E. (eds.) Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016. pp. 434–447. ACM (2016)
55. Lynch, N.A.: Fast allocation of nearby resources in a distributed system. In: Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing. p. 70–81. STOC '80, Association for Computing Machinery, New York, NY, USA (1980)

56. Marlow, S.: Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming. " O'Reilly Media, Inc." (2013)
57. Militão, F., Aldrich, J., Caires, L.: Aliasing control with view-based typestate. In: Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs. pp. 1–7 (2010)
58. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. ACM Transactions on Programming Languages and Systems (TOPLAS) **10**(3), 470–502 (1988)
59. Mostrous, D., Vasconcelos, V.T.: Affine Sessions. In: Proc. of COORDINATION 2014. LNCS, vol. 8459, pp. 115–130. Springer (2014)
60. Nanevski, A., Morrisett, J.G., Birkedal, L.: Hoare type theory, polymorphism and separation. J. Funct. Program. **18**(5-6), 865–911 (2008)
61. O'Hearn, P.W., Reynolds, J.C.: From Algol to polymorphic linear lambda-calculus. J. ACM **47**(1), 167–223 (2000)
62. Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations and observational equivalences for session-based concurrency. Information and Computation **239**, 254–302 (2014)
63. Pfenning, F.: Structural cut elimination. In: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science. p. 156. LICS '95, IEEE Computer Society, USA (1995)
64. Qian, Z., Kavvos, G., Birkedal, L.: Client-server sessions in linear logic. Proceedings of the ACM on Programming Languages **5**(ICFP), 1–31 (2021)
65. Rocha, P.: A Logical Foundation for Session-Based Concurrent Computation. Ph.D. thesis, NOVA School of Science and Technology (July 2022)
66. Rocha, P., Caires, L.: A Propositions-as-Types System for Shared State. Tech. rep., NOVA Laboratory for Computer Science and Informatics (06 2021)
67. Rocha, P., Caires, L.: Propositions-as-types and shared state. Proceedings of the ACM on Programming Languages **5**(ICFP), 1–30 (2021)
68. Rocha, P., Caires, L.: Safe ssession-based concurrency with shared linear state (artifact) (January 2023). https://doi.org/10.5281/zenodo.7506064
69. Sangiorgi, D.: Termination of processes. Math. Struct. in Comp. Sci. **16**(1), 1–39 (2006)
70. Sangiorgi, D., Walker, D.: PI-Calculus: A Theory of Mobile Processes. Cambridge University Press, USA (2001)
71. Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming. p. 161–172. PPDP '11, Association for Computing Machinery, New York, NY, USA (2011)
72. Toninho, B., Caires, L., Pfenning, F.: Corecursion and non-divergence in session-typed processes. In: Maffei, M., Tuosto, E. (eds.) TGC 2014. Lecture Notes in Computer Science, vol. 8902, pp. 159–175. Springer (2014)
73. Toninho, B., Caires, L., Pfenning, F.: Corecursion and non-divergence in session-typed processes. In: International Symposium on Trustworthy Global Computing. pp. 159–175. Springer (2014)
74. Toninho, B., Yoshida, N.: On polymorphic sessions and functions: A tale of two (fully abstract) encodings. ACM Trans. Program. Lang. Syst. **43**(2) (Jun 2021)
75. Tov, J.A., Pucella, R.: Practical Affine Types. In: POPL 2011. pp. 447–458 (2011)
76. Vieira, H.T., Vasconcelos, V.T.: Typing progress in communication-centred systems. In: International Conference on Coordination Languages and Models. pp. 236–250. Springer (2013)

77. Voinea, A.L., Dardha, O., Gay, S.J.: Resource sharing via capability-based multiparty session types. In: International Conference on Integrated Formal Methods. pp. 437–455. Springer (2019)
78. Wadler, P.: Linear types can change the world! In: Broy, M. (ed.) Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, 1990. p. 561. North-Holland (1990)
79. Wadler, P.: Recursive types for free (1990)
80. Wadler, P.: Propositions as sessions. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. p. 273–286. ICFP '12, Association for Computing Machinery, New York, NY, USA (2012)
81. Wadler, P.: Propositions as Sessions. Journal of Functional Programming **24**(2-3), 384–418 (2014)
82. Wadler, P.: Propositions as types. Communications of the ACM **58**(12), 75–84 (2015)
83. Yoshida, N., Berger, M., Honda, K.: Strong normalisation in the π-calculus. Information and Computation **191**(2), 145–202 (2004)

Bunched Fuzz: Sensitivity for Vector Metrics

june wunder[1](✉), Arthur Azevedo de Amorim[3], Patrick Baillot[2], and Marco Gaboardi[1]

[1] Boston University, Boston, USA
jwunder@bu.edu

[2] Univ. Lille, CNRS, Inria, Centrale Lille, UMR9189 CRIStAL, F-59000 Lille, France

[3] Rochester Institute of Technology, Rochester, USA

Abstract. *Program sensitivity* measures the distance between the outputs of a program when run on two related inputs. This notion, which plays a key role in areas such as data privacy and optimization, has been the focus of several program analysis techniques introduced in recent years. Among the most successful ones, we can highlight type systems inspired by linear logic, as pioneered by Reed and Pierce in the Fuzz programming language. In Fuzz, each type is equipped with its own distance, and sensitivity analysis boils down to type checking. In particular, Fuzz features two product types, corresponding to two different notions of distance: the *tensor product* combines the distances of each component by *adding* them, while the *with product* takes their *maximum*.

In this work, we show that these products can be generalized to arbitrary L^p *distances*, metrics that are often used in privacy and optimization. The original Fuzz products, tensor and with, correspond to the special cases L^1 and L^∞. To ease the handling of such products, we extend the Fuzz type system with *bunches*—as in the logic of bunched implications—where the distances of different groups of variables can be combined using different L^p distances. We show that our extension can be used to reason about quantitative properties of probabilistic programs.

1 Introduction

When developing a data-driven application, we often need to analyze its *sensitivity*, or *robustness*, a measure of how its outputs can be affected by varying its inputs. For example, to analyze the privacy guarantees of a program, we might consider what happens when we include the data of one individual in its inputs [11]. When analyzing the stability of a machine-learning algorithm, we might consider what happens when we modify one sample in the training set [7].

Such applications have spurred the development of several techniques to reason about program sensitivity [23,9]. One successful approach is based on linear-like [14] type systems, as pioneered in Reed and Pierce's *Fuzz* language [23].

The basic idea behind Fuzz is to use typing judgments to track the sensitivity of a program with respect to each variable. Each type comes equipped with a notion of distance, and the typing rules explain how to update variable sensitivities for each operation. Because different distances yield different sensitivity analyses, it is often useful to endow a set of values with different distances, which

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 451–478, 2023.
https://doi.org/10.1007/978-3-031-30044-8_17

leads to different Fuzz types. For example, like linear logic, Fuzz has two notions of products: the tensor product $\otimes$ and the Cartesian product & (with). The first one is equipped with the L^1 (or Manhattan) distance, where the distance between two pairs is computed by *adding* the distances between the corresponding components. The second one is equipped with the L^∞ (or Chebyshev) distance, where the component distances are combined by taking their *maximum*.

The reason for focusing on these two product types is that they play a key role in differential privacy [11], a rigorous notion of privacy that was the motivating application behind the original Fuzz design. However, we could also consider equipping pairs with more general L^p *distances*, which interpolate between the L^1 and L^∞ and are extensively used in convex optimization [8], information theory [10] and statistics [15]. Indeed, other type systems for differential privacy inspired by Fuzz [20] include types for vectors and matrices under the L^2 distance, which are required to use the Gaussian mechanism, one of the popular building blocks of differential privacy. Supporting more general L^p metrics would allow us to capture even more such building blocks [17,1], which would enable further exploration of the tradeoffs between differential privacy and accuracy.

In this paper, we extend these approaches and show that Fuzz can be enriched with a family of tensor products $\otimes_p$, for $1 \leq p \leq \infty$. These tensor products are equipped with the L^p distance, the original Fuzz products $\otimes$ and & corresponding to the special cases $\otimes_1$ and $\otimes_\infty$. Moreover, each connective $\otimes_p$ is equipped with a corresponding "linear implication" $\multimap_p$, unlike previous related systems where such an implication only exists for $p = 1$. Following prior work [4,3], we give to our extension a semantics in terms of non-expansive functions, except that the presence of the implications $\multimap_p$ forces us to equip input and output spaces with more general distances where the triangle inequality need not hold.

A novelty of our approach is that, to support the handling of such products, we generalize Fuzz environments to *bunches*, where each L^p distance comes with its own context former. Thus, we call our type system Bunched Fuzz. This system, inspired by languages derived from the logic of Bunched Implications (BI) [22] (e.g. [21]), highlights differences between the original Fuzz design and linear logic—for example, products distribute over sums in Fuzz and BI, but not in linear logic. While similar indexed products and function spaces have also appeared in the literature, particularly in works on categorical grammars [19], here they are employed to reason about vector distances and function sensitivity.

While designing Bunched Fuzz, one of our goals was to use sensitivity to reason about randomized algorithms. In the original Fuzz, probability distributions are equipped with the *max divergence* distance, which can be used to state differential privacy as a sensitivity property [23]. Subsequent work has shown how Fuzz can also accommodate other distances over probability distributions [3]. However, such additions required variants of *graded monads*, which express the distance between distributions using indices (i.e. grades) on the monadic type of distributions over their *results*, as opposed to sensitivity indices on their *inputs*, as it was done in the original Fuzz. In particular, this makes it more difficult to reason about distances separately with respect to each input. Thanks to bunches, however, we can incorporate these composition principles more naturally. For example, Bunched Fuzz can reason about the Hellinger distance on distributions without the need for output grading, as was done in prior systems [3].

We will also see that, by allowing arbitrary L^p norms, we can generalize prior case studies that were verified in Fuzz and obtain more general methods for reasoning about differential privacy (Section 5). Consider the L^p mechanism [1,17], which adds noise to the result of a query whose sensitivity is measured in the L^p norm. Since Fuzz does not have the means to analyze such a sensitivity measure, it cannot implement the L^p mechanism; Bunched Fuzz, however, can analyze such a measure, and thus allows for a simple implementation in terms of the exponential mechanism. Such a mechanism, in turn, can be used to implement a variant of a gradient descent algorithm that works under the L^p norm, generalizing an earlier version that was biased towards the L^1 norm [25]. Summarizing, our contributions are:

– We introduce Bunched Fuzz, an extension of Fuzz with types for general L^p distances: we add type constructors of the form $\otimes_p$ (for $1 \leq p \leq \infty$) for pairs under the L^p distance along with constructors of the form $\multimap_p$ for their corresponding function spaces. To support the handling of such types, we generalize Fuzz typing contexts to *bunches of variable assignments*.
– We give a denotational semantics for Bunched Fuzz by interpreting programs as non-expansive functions over spaces built on L^p distances.
– We show that Bunched Fuzz can support types for probability distributions for which the sampling primitive, which enables the composition of probabilistic programs, is compatible with L^p distances.
– We show a range of examples of programs that can be written in Bunched Fuzz. Notably, we show that Bunched Fuzz can support reasoning about the Hellinger distance without the need for grading, and we show generalizations of several examples from the differential privacy literature.

Check the full version of this paper for more technical details [26].

2 Background

2.1 Metrics and Sensitivity

To discuss sensitivity, we first need a notion of distance. We call *extended pseudosemimetric space* a pair $X = (|X|, d_X)$ consisting of a carrier set $|X|$ and an *extended pseudosemimetric* $d_X : |X|^2 \to \mathbb{R}^{\geq 0}_{\infty}$, which is a function satisfying, for all $x, y \in |X|$, $d_X(x, x) = 0$ and $d_X(x, y) = d_X(y, x)$. This relaxes the standard notion of metric space in a few respects. First, the distance between two points can be infinite, hence the *extended*. Second, different points can be at distance zero, hence the *pseudo*. Finally, we do not require the *triangular inequality*:

$$d_X(x, y) \leq d_X(x, z) + d_X(z, y), \tag{1}$$

hence the *semi*. We focus on extended pseudosemimetrics because they support constructions that true metrics do not. In particular, they make it possible to scale the distance of a space by ∞ and enable more general function spaces. However, to simplify the terminology, we will drop the "extended pseudosemi" prefix in the rest of the paper, and speak solely of metric spaces. In some occasions, we might speak of a *proper metric space*, by which we mean a space where the triangle inequality *does* hold (but not necessarily the other two requirements that are missing compared to the traditional definition of metric space).

Given a function $f : X \to Y$ on metric spaces, we say that it is s-sensitive, for s in $\mathbb{R}^{\geq 0}_{\infty}$, if we have, for all $x_1, x_2 \in X$, $d_Y(f(x_1), f(x_2)) \leq s \cdot d_X(x_1, x_2)$, where we extend addition and multiplication to $\mathbb{R}^{\geq 0}_{\infty}$ by setting $\infty \cdot s = s \cdot \infty = \infty$. We also say that f is *s-Lipschitz continuous*, though the traditional definition of Lipschitz continuity assumes $s \neq \infty$. If a function is s-sensitive, then it is also s'-sensitive for every $s' \geq s$. Every function of type $X \to Y$ is ∞-sensitive. If a function is 1-sensitive, we also say that f is non-expansive. We use $X \multimap Y$ to denote the set of such non-expansive functions. The identity function is always non-expansive, and non-expansive functions are closed under composition. Thus, metric spaces and non-expansive functions form a category, denoted Met.

2.2 Distances for Differential Privacy

Among many applications, sensitivity is a useful notion because it provides a convenient language for analyzing the privacy guarantees of algorithms—specifically, in the framework of differential privacy [11]. Differential privacy is a technique for protecting the privacy of individuals in a database by blurring the results of a query to the database with random noise. The noise is calibrated so that each individual has a small influence on the probability of observing each outcome (while ideally guaranteeing that the result of the query is still useful).

Formally, suppose that we have some set of databases db equipped with a metric. This metric roughly measures how many rows differ between two databases, though the exact definition can vary. Let $f : \mathsf{db} \to DX$ be a randomized database query, which maps a database to a discrete probability distribution over the set of outcomes X. We say that f is *ϵ-differentially private* if it is an ϵ-sensitive function from db to DX, where the set of distributions DX is equipped with the following distance, sometimes known as the *max divergence*:

$$\mathsf{MD}_X(\mu_1, \mu_2) = \sum_{x \in X} \ln \left| \frac{\mu_1(x)}{\mu_2(x)} \right|. \tag{2}$$

(Here, we stipulate that $\ln |0/0| = 0$ and $\ln |p/0| = \ln |0/p| = \infty$ for $p \neq 0$.)

To understand this definition, suppose that D_1 and D_2 are two databases at distance 1—for instance, because they differ with respect to the data of a single individual. If f is ϵ-differentially private, the above definition implies that $f(D_1)$ and $f(D_2)$ are at most ϵ apart. When ϵ is large, the probabilities of each outcome in the result distributions can vary widely. This means that, by simply observing one output of f, we might be able to guess with good confidence which of the databases D_1 or D_2 was used to produce that output. Conversely, if ϵ is small, it is hard to tell which database was used because the output probabilities will be close. For this reason, it is common to view ϵ as a *privacy loss*—the larger it is, the more privacy we are giving up to reveal the output of f.

Besides providing a strong privacy guarantee, this formulation of closeness for distributions provides two important properties. First, we can *compose* differentially private algorithms without ruining their privacy guarantee. Note that DX forms a monad, where the return and bind operations are given as follows:

$$\eta(x) = y \mapsto \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$$f^\dagger(\mu) = y \mapsto \sum_{x \in X} \mu(x) \cdot f(x)(y). \tag{4}$$

Intuitively, the return operation produces a deterministic distribution, whereas bind samples an element x from μ and computes $f(x)$. When composing differentially private algorithms, their privacy loss can be soundly added together:

Theorem 1. *Suppose that* $f : \mathsf{db} \to DX$ *is* ϵ_1*-differentially private and that* $g : \mathsf{db} \to X \to DY$ *is such that the mapping* $\delta \to g(\delta)(x)$ *is* ϵ_2*-differentially private for every* x*. Then the composite* $h : \mathsf{db} \to DY$ *defined as* $h(\delta) = g(\delta)^\dagger(f(\delta))$ *is* $(\epsilon_1 + \epsilon_2)$*-differentially private.*

The other reason why the privacy metric is useful is that it supports many building blocks for differential privacy. Of particular interest is the *Laplace mechanism*, which blurs a numeric result with noise drawn from the two-sided Laplace distribution. If $x \in \mathbb{R}$, let $\mathcal{L}(x)$ be the distribution with density[4] $y \mapsto \frac{1}{2}e^{-|x-y|}$.

Theorem 2. *The mechanism* $\mathcal{L}$ *is a non-expansive function of type* $\mathbb{R} \to D\mathbb{R}$.[5]

Thus, to define an ϵ-differentially private numeric query on a database, it suffices to define an ϵ-sensitive, deterministic numeric query, and then blur its result with Laplace noise. Differential privacy follows from the composition principles for sensitivity. This reasoning is justified by the fact that the Laplace mechanism adds noise proportional to the sensitivity of the numeric query in L^1 distance.

2.3 Sensitivity as a Resource

Because differential privacy is a sensitivity property, techniques for analyzing the sensitivity of programs can also be used to analyze their privacy guarantees. One particularly successful approach in this space is rooted in type systems inspired by linear logic, as pioneered by Reed and Pierce in the Fuzz programming language [16,23]. At its core, Fuzz is just a type system for tracking sensitivity.

[4] We use here a Laplace distribution with scale 1.

[5] The definitions do not quite match up our setting, since $\mathcal{L}$ is a continuous, and not discrete distribution. The result can be put on firm footing by working with a discretized version of the Laplace distribution [12].

Typing judgments are similar to common functional programming languages, but variable declarations are of the form $x_i :_{r_i} \tau_i$: $x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n \vdash e : \sigma$. The annotations $r_i \in \mathbb{R}_\infty^{\geq 0}$ are *sensitivity indices*, whose purpose is to track the effect that changes to the program input can have on its output: if we have two substitutions γ and γ' for the variables x_i, then the *metric preservation* property of the Fuzz type system guarantees that

$$d(e[\gamma/\vec{x}], e[\gamma'/\vec{x}]) \leq \sum_i r_i \cdot d(\gamma(x_i), \gamma'(x_i)), \tag{5}$$

where the metrics d are computed based on the type of each expression and value. This means that we can bound the distance on the results of the two runs of e by adding up the distances of the inputs scaled by their corresponding sensitivities. When this bound is finite, the definition of the metrics guarantees that the two runs have the same termination behavior. When $r_i = \infty$, the above inequality provides no guarantees if the value of x_i varies.

Fuzz includes data types commonly found in functional programming languages, such as numbers, products, tagged unions, recursive types and functions. The typing rules of the language explain how the sensitivities of each variable must be updated to compute each operation. The simplest typing rule says that, in order to use a variable, its declared sensitivity must be greater than 1:

$$\frac{r \geq 1}{\Gamma, x :_r \tau, \Delta \vdash x : \tau}$$

As a more interesting example, to construct a pair (e_1, e_2), the following rule says that we need to add the sensitivities of the corresponding contexts:

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : \tau_1 \otimes \tau_2}.$$

This behavior is a result of the distance of the tensor type $\otimes$: the distance between two pairs in $\tau_1 \otimes \tau_2$ is the result of adding the distances between the first and second components; therefore, the sensitivity of each variable for the entire expression is the sum of the sensitivities for each component. In this sense, sensitivities in Fuzz behave like a resource that must be distributed across all variable uses in a program. For the sake of analogy, we might compare this treatment to how fractional permissions work in separation logic: the predicate $l \mapsto_q x$ indicates that we own a fraction $q \in [0,1]$ of a resource stating that l points to x. If $q = q_1 + q_2$, we can split this predicate as $l \mapsto_{q_1} x * l \mapsto_{q_2} x$, allowing us to distribute this resource between different threads.

The distance on $\otimes$ corresponds to the sum in the upper bound in the statement of metric preservation (Equation (5)). This distance is useful because it is the one that yields good composition principles for differential privacy. This can be seen in the typing rule for sampling from a probabilistic distribution:

$$\frac{\Gamma \vdash e_1 : \bigcirc\tau \qquad \Delta, x :_r \tau \vdash e_2 : \bigcirc\sigma}{\Gamma + \Delta \vdash \textbf{mlet } x = e_1 \textbf{ in } e_2 : \bigcirc\sigma}$$

Here, $\bigcirc\tau$ denotes the type of probability distributions over values of type τ. This operation samples a value x from the distribution e_1 and uses this value to compute the distribution e_2. We can justify the soundness of this rule by reducing it to Theorem 1: the addition on contexts corresponds to the fact that the privacy loss of a program degrades linearly under composition.

Besides the tensor product $\otimes$, Fuzz also features a *with product* &, where the distances between components are combined by taking their maximum. This leads to a different typing rule for & pairs, which does not add up the sensitivities:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \mathbin{\&} \tau_2}$$

If we compare these rules for pairs, we see a clear analogy with linear logic: $\otimes$ requires us to combine contexts, whereas & allows us to share them. Fuzz's elimination rules for products continue to borrow from linear logic: deconstructing a tensor gives both elements but deconstructing a with product returns only one.

$$\frac{\Gamma \vdash e : \tau_1 \otimes \tau_2 \qquad \Delta, x :_r \tau_1, y :_r \tau_2 \vdash e' : \tau'}{\Delta + r\Gamma \vdash \mathbf{let}\ (x, y) = e\ \mathbf{in}\ e' : \tau'} \qquad \frac{\Gamma \vdash e : \tau_1 \mathbin{\&} \tau_2}{\Gamma \vdash \pi_i\ e : \tau_i}$$

This partly explains why the connectives' distances involve addition and maximum. When using a tensor product, both elements can affect how much the output can vary, so both elements must be considered. (Note that Fuzz is an affine type system: we are free to ignore one of the product's components, and thus we can write projection functions out of a tensor product.) When projecting out of a with product, only one of the elements will affect the program's output, so we only need to consider the component that yields the maximum distance.

Fuzz uses the $!_s$ type for managing sensitivities. Intuitively, $!_s\tau$ behaves like τ, but with the distances scaled by s; when $s = \infty$, this means that different points are infinitely apart. The introduction rule scales the sensitivities of variables in the environment. This can be used in conjunction with the elimination rule to propagate the sensitivity out of the type and into the environment.

$$\frac{\Gamma \vdash e : \tau}{s\Gamma \vdash !e : !_s\tau} \qquad \frac{\Gamma \vdash e : !_s\tau \qquad \Delta, x :_{rs} \tau \vdash e' : \tau'}{\Delta + r\Gamma \vdash \mathbf{let}\ !x = e\ \mathbf{in}\ e' : \tau'}$$

Finally, the rules for the linear implication $\multimap$ are similar to the ones from linear logic, but adjusted to account for sensitivities.

$$\frac{\Gamma, x :_1 \tau \vdash e : \sigma}{\Gamma \vdash \lambda x.e : \tau \multimap \sigma} \qquad \frac{\Gamma \vdash e : \tau \multimap \sigma \qquad \Delta \vdash e' : \tau}{\Gamma + \Delta \vdash e\ e' : \sigma}$$

To introduce the linear implication $\multimap$, the bound variable needs to have sensitivity 1. When eliminating $\multimap$, the environments need to be added. In categorical language, addition, which is also present in the metric for $\otimes$, is connected to the fact that there is an adjunction between the functors $X \otimes (-)$ and $X \multimap (-)$.

2.4 L^p distances

The L^1 and L^∞ distances are instances of a more general family of L^p distances (for $p \in \mathbb{R}^{\geq 1}_{\infty}$).[6] Given a sequence of distances $\vec{x} = (x_1, \ldots, x_n) \in (\mathbb{R}^{\geq 0}_{\infty})^n$, we first define the L^p *pseudonorm*[7] as follows: $||\vec{x}||_p = (\Sigma_{i=1}^n x_i^p)^{1/p}$. This definition makes sense whenever the distances x_i and p are finite. When $p = \infty$, we define the right-hand side as the limit $\max_{i=1}^n x_i$. When $x_i = \infty$ for some i, we define the right-hand side as ∞. We have the following classical properties:

Proposition 1 (Hölder inequality). *For all $p, q \geq 1$ such that $\frac{1}{p} + \frac{1}{q} = 1$, and for all $\vec{x}$, $\vec{y} \in (\mathbb{R}^{\geq 0}_{\infty})^n$, we have: $\Sigma_{i=1}^n x_i y_i \leq ||\vec{x}||_p ||\vec{y}||_q$.*
For $p = 2$, $q = 2$, this is the Cauchy-Schwarz inequality: $\Sigma_{i=1}^n x_i y_i \leq ||\vec{x}||_2 ||\vec{y}||_2$.

Proposition 2. *For $1 \leq p \leq q$ we have, for $\vec{x} \in (\mathbb{R}^{\geq 0}_{\infty})^n$:*

$$||\vec{x}||_q \leq ||\vec{x}||_p \tag{6}$$

$$||\vec{x}||_p \leq n^{\frac{1}{p} - \frac{1}{q}} ||\vec{x}||_q \tag{7}$$

$$||\vec{x}||_2 \leq ||\vec{x}||_1 \leq \sqrt{n}\, ||\vec{x}||_2 \tag{8}$$

The L^p pseudonorms yield distances on tuples. More precisely, suppose that $(X_i)_{1 \leq i \leq n}$ are metric spaces. The following defines a metric on $X = X_1 \times \cdots \times X_n$:

$$d_p(\vec{x}, \vec{x}') = ||(d_{X_1}(x_1, x_1'), \ldots, d_{X_n}(x_n, x_n'))||_p$$

Proposition 3. *For $1 \leq p \leq q$ we have, for $\vec{x}, \vec{x'} \in X_1 \times \cdots \times X_n$:*

$$d_q(\vec{x}, \vec{x'}) \leq d_p(\vec{x}, \vec{x'}) \leq n^{\frac{1}{p} - \frac{1}{q}} d_q(\vec{x}, \vec{x'}) \tag{9}$$

$$d_2(\vec{x}, \vec{x'}) \leq d_1(\vec{x}, \vec{x'}) \leq \sqrt{n}\, d_2(\vec{x}, \vec{x'}) \tag{10}$$

3 Bunched Fuzz: Programming with L^p Distances

As we discussed earlier, the L^1 distance is not the only distance on products with useful applications. In the context of differential privacy, for example, the L^2 distance is used to measure the sensitivity of queries when employing the Gaussian mechanism, a method for private data release that sanitizes data by adding Gaussian noise instead of Laplacian noise.[8]

It is possible to extend a Fuzz-like analysis with L^2 distances by adding primitive types and combinators for vectors. This was done, for instance, in

[6] The L^p distances can be defined with $p \geq 0$ but for simplicity of our treatment we will only consider $p \geq 1$.

[7] "pseudo-" because it can be infinite.

[8] Technically, the Gaussian mechanism is used to achieve a relaxation of differential privacy known as *approximate*, or (ϵ, δ)-differential privacy. Though this notion cannot be analyzed directly by classical verification techniques for differential privacy, it can be handled by recent extensions of Fuzz [3,20].

the Duet language [20], which provides the Gaussian mechanism as one of the primitives for differential privacy. Such an extension can help verify a wide class of algorithms that manipulate vectors in a homogeneous fashion, but it makes it awkward to express programs that require finer grained access to vectors.

To illustrate this point, suppose that we have a non-expansive function $f : \mathbb{R}^2 \to \mathbb{R}$, where the domain carries the L^2 metric. Consider the mapping

$$g(x, y) = f(2x, y) + f(2y, x).$$

How would we analyze the sensitivity of g? We cannot translate such a program directly into a system like Duet, since it does not allow us to manipulate L^2 vectors at the level of individual components. However, we could rewrite the definition of g to use matrix operations, which could be easily incorporated in a variant of Duet. Specifically, consider the following definition:

$$g(\vec{x}) = f\left(\begin{bmatrix}2 & 0\\0 & 1\end{bmatrix}\vec{x}\right) + f\left(\begin{bmatrix}0 & 2\\1 & 0\end{bmatrix}\vec{x}\right).$$

The L^2 sensitivity of a linear transformation $\vec{x} \mapsto M\vec{x}$ can be easily computed if we know the coefficients of the matrix M. Note that

$$\begin{aligned} d(M\vec{x}, M\vec{y}) = ||M\vec{x} - M\vec{y}||_2 = ||M(\vec{x} - \vec{y})||_2 &= \frac{||M(\vec{x} - \vec{y})||_2}{||\vec{x} - \vec{y}||_2}||\vec{x} - \vec{y}||_2 \\ &\leq \left(\sup_{\vec{z}} \frac{||M\vec{z}||_2}{||\vec{z}||_2}\right) d(\vec{x}, \vec{y}). \end{aligned}$$

The quantity $\sup_{\vec{z}} ||M\vec{z}||_2/||\vec{z}||_2$, known as the *operator norm* of M, gives the precise sensitivity of the above operation, and can be computed by standard algorithms from linear algebra. In the case of g, both matrices have a norm of 2. This means that we can analyze the sensitivity of g compositionally, as in Fuzz: addition is 1-sensitive in each variable, so we just have to sum the sensitivities of $\vec{x}$ in each argument, yielding a combined sensitivity of 4. Unfortunately, this method of combining the sensitivities of each argument is too coarse when reasoning with L^p distances, which leads to an imprecise analysis. To obtain a better bound, we can reason informally as follows. First, take $M = \begin{bmatrix}2 & 0 & 0 & 1\\0 & 1 & 2 & 0\end{bmatrix}^T$. We can compute the operator norm of M directly:

$$||M|| = \sup_{x,y} \frac{\sqrt{2^2x^2 + y^2 + 2^2y^2 + x^2}}{\sqrt{x^2 + y^2}} = \sup_{x,y} \frac{\sqrt{5(x^2 + y^2)}}{\sqrt{x^2 + y^2}} = \sqrt{5},$$

which implies that M is a $\sqrt{5}$-sensitive function of type $\mathbb{R}^2 \to \mathbb{R}^4 \cong \mathbb{R}^2 \times \mathbb{R}^2$. Moreover, thanks to Proposition 3, we can view addition $(+)$ as a $\sqrt{2}$-sensitive operator of type $\mathbb{R}^2 \to \mathbb{R}$, since

$$d_{\mathbb{R}}(x_1 + x_2, y_1 + y_2) \leq d_{\mathbb{R}}(x_1 - y_1) + d_{\mathbb{R}}(x_2 - y_2) = d_1(\vec{x}, \vec{y}) \leq \sqrt{2}d_2(\vec{x}, \vec{y}).$$

$$
\begin{aligned}
\tau, \sigma, \rho &::= 1 \mid \mathbb{R} \mid !_s\tau \mid \bigcirc_P \tau \mid \bigcirc_H \tau \mid \tau \multimap_p \sigma \mid \tau \otimes_p \sigma \mid \tau \oplus \sigma \qquad (p \in \mathbb{R}^{\geq 1}_{\infty}, s \in \mathbb{R}^{\geq 0}_{\infty}) \\
e &::= x \mid r \in \mathbb{R} \mid () \mid \lambda x.e \mid e\, e \mid (e, e) \mid \mathbf{let}\ (x, y) = e\ \mathbf{in}\ e \\
&\mid \mathbf{inj}_i e \mid (\mathbf{case}\ e\ \mathbf{of}\ x.\ e \mid y.\ e) \mid !e \mid \mathbf{let}\ !x = e\ \mathbf{in}\ e \\
&\mid \mathbf{mlet}\ x = e\ \mathbf{in}\ e \mid \mathbf{return}\ e \mid \cdots
\end{aligned}
$$

Fig. 1. Types and terms in Bunched Fuzz

Thus, by rewriting the definition of g as $(+) \circ (f \times f) \circ M$, where $f \times f : \mathbb{R}^4 \cong \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R} \times \mathbb{R}$ denotes the application of f in parallel, we can compute the sensitivity of g by multiplying the sensitivity of each stage, as $\sqrt{2} \times 1 \times \sqrt{5} = \sqrt{10} \approx 3.16$, which is strictly better than the previous bound.

Naturally, we could further extend Fuzz or Duet with primitives for internalizing this reasoning, but it would be preferable to use the original definition of g and automate the low-level reasoning about distances. In this section, we demonstrate how this can be done via Bunched Fuzz, a language that refines Fuzz by incorporating more general distances in its typing environments. Rather assuming that input distances are always combined by addition, or the L^1 distance, Bunched Fuzz allows them to be combined with arbitrary L^p distances. This refinement allows us to analyze different components of a vector as individual variables, but also to split the sensitivity of these variables while accounting for their corresponding vector distances. In the remaining of this section, we present the syntax and type system of Bunched Fuzz, highlighting the main differences with respect to the original Fuzz design. Later, in Section 4, we will give a semantics to this language in terms of metric spaces, following prior work [3].

Types and Terms Figure 1 presents the grammar of types and the main term formers of Bunched Fuzz. They are similar to their Fuzz counterparts; in particular, there are types for real numbers, products, sums, functions, and a unit type. The main novelty is in the product type $\tau \otimes_p \sigma$, which combines the metrics of each component using the L^p distance (cf. Section 2.4). The types $\tau \otimes_1 \sigma$ and $\tau \otimes_\infty \sigma$ subsume the types $\tau \otimes \sigma$ and $\tau \,\&\, \sigma$ in the original Fuzz language. Note that there is no term constructor or destructor for the Fuzz type &, since it is subsumed by $\otimes_\infty$. The type $\tau \multimap_p \sigma$ represents non-expansive functions endowed with a metric that is compatible with the L^p metric, in that currying works (cf. Section 5). We will sometimes write $\otimes$ for $\otimes_1$ and $\multimap$ for $\multimap_1$.

Another novelty with respect to Fuzz is that there are two constructors for probability distributions, $\bigcirc_P$ and $\bigcirc_H$. The first one carries the original Fuzz privacy metric, while the second one carries the Hellinger distance. As we will see shortly, the composition principle for the Hellinger distance uses a contraction operator for the L^2 distance, which was not available in the original Fuzz design. Both distribution types feature term constructors **mlet** and **return** for sampling from a distribution and for injecting values into distributions. To simplify the notation, we do not use separate versions of these term formers for each type.

Bunches Before describing its type system, we need to talk about how typing environments are handled in Bunched Fuzz. In the spirit of bunched logics, environments are bunches defined with the following grammar:

$$\Gamma, \Delta ::= \cdot \mid [x : \tau]_s \mid \Gamma \,,_p \Delta$$

The empty environment is denoted as $\cdot$. The form $[x : \tau]_s$ states that the variable x has type τ and sensitivity s. The form $\Gamma \,,_p \Delta$ denotes the concatenation of Γ and Δ, which is only defined when the two bind disjoint sets of variables. As we will see in Section 4, bunches will be interpreted as metric spaces, and the p index denote which L^p metric we will use to combine the metrics of Γ and Δ.

The type system features several operations and relations on bunches, which are summarized in Figure 2. We write $\Gamma \leftrightsquigarrow \Gamma'$ to indicate that we can obtain Γ' by rearranging commas up to associativity and commutativity, and by treating the empty environment as an identity element; Figure 2 has a precise definition. Observe that associativity only holds for equal values of p. This operation will be used to state a permutation rule for the type system of Bunched Fuzz.

Like in Fuzz, environments have a scaling operation $s\Gamma$ which scales all sensitivities in the bunch by s. For example,

$$s([x : \tau]_{r_1} \,,_p [y : \sigma]_{r_2}) = ([x : \tau]_{s \cdot r_1} \,,_p [y : \sigma]_{s \cdot r_2}).$$

The exact definition of scaling in such graded languages is subtle, since minor variations can quickly lead to unsoundness. The definition we are using ($\infty \cdot 0 = 0 \cdot \infty = \infty$), which goes back to prior work [3], is sound, but imprecise, since it leads to too many variables being marked as ∞-sensitive. It would also be possible to have a more precise variant that uses a non-commutative definition of multiplication on distances [4], but we keep the current formulation for simplicity. (For a more thorough discussion on these choices and their tradeoffs, see the "Zero and Infinity" example in Appendix B of the full version [26] of this paper.)

In the original Fuzz type system, rules with several premises usually have their environments combined by adding sensitivities pointwise, which corresponds to a use of the L^1 metric. In Bunched Fuzz, we have instead a family of contraction operations $\mathit{Contr}(p, \Gamma, \Delta)$ for combining environments, one for each L^p metric. Contraction only makes sense if Γ and Δ differ only in sensitivities and variable names, but have the same structure otherwise. We write this relation as $\Gamma \approx \Delta$. When contracting two leaves, sensitivities are combined using the L^p norm, while keeping variable names from the left bunch.

Unlike Fuzz, where contraction is implicit in rules with multiple premises, Bunched Fuzz has a separate, explicit contraction typing rule. The rule will be stated using the *vars* function, which lists all variables in a bunch.

Type System Our type system is similar to the one of Fuzz, but adapted to use bunched environments. The typing rules are displayed on Figure 3. For example, in the $\otimes$I rule, notice that the p on the tensor type is carried over to the bunch in the resulting environment. Similarly, in the $\multimap$I rule, the value of p that annotates the bunch in the premise is carried over to the $\multimap_p$ in the conclusion.

$$\begin{array}{rlrl}
vars(\cdot) = [] & & \cdot \approx \cdot & \\
vars([x:\tau]_s) = [x] & & [x:\tau]_s \approx [y:\sigma]_r & \text{if } \tau = \sigma \\
vars((\Gamma_1,_p \Gamma_2)) = vars(\Gamma_1) +\!\!+\, vars(\Gamma_2) & & \Gamma_1 ,_p \Gamma_2 \approx \Delta_1,_q \Delta_2 & \text{if } p = q \wedge \Gamma_i \approx \Delta_i
\end{array}$$

$$\begin{array}{rcll}
\Gamma \leftrightsquigarrow \Delta & \text{if } \Gamma = \Delta & & \\
\Gamma \leftrightsquigarrow \cdot,_p \Delta & \text{if } \Gamma \leftrightsquigarrow \Delta & & \\
\Gamma \leftrightsquigarrow \Delta,_p \cdot & \text{if } \Gamma \leftrightsquigarrow \Delta & & s\,\cdot = \cdot \\
\Gamma_1,_p \Gamma_2 \leftrightsquigarrow \Delta_1,_p \Delta_2 & \text{if } \Gamma_i \leftrightsquigarrow \Delta_i & & s\,[\tau]_r = [\tau]_{s\cdot r} \\
\Gamma_1,_p \Gamma_2 \leftrightsquigarrow \Delta_2,_p \Delta_1 & \text{if } \Gamma_i \leftrightsquigarrow \Delta_i & & s\,(\Gamma ,_p \Delta) = s\Gamma ,_p s\Delta \\
\Gamma_1,_p (\Gamma_2,_p \Gamma_3) \leftrightsquigarrow (\Delta_1,_p \Delta_2),_p \Delta_3 & \text{if } \Gamma_i \leftrightsquigarrow \Delta_i & & \\
\Gamma_2 \leftrightsquigarrow \Gamma_1 & \text{if } \Gamma_1 \leftrightsquigarrow \Gamma_2 & &
\end{array}$$

$$\begin{aligned}
c(p,q) &= \begin{cases} 1 & \text{if } p = \infty \\ 2^{\left|\frac{1}{q}-\frac{1}{p}\right|} & \text{otherwise} \end{cases} \\
Contr(p,\cdot,\cdot) &= \cdot \\
Contr(p,[x:\tau]_s,[y:\tau]_r) &= [x:\tau]_{\sqrt[p]{s^p+r^p}} \\
Contr(p,(\Gamma_1,_q \Gamma_2),(\Delta_1,_q \Delta_2)) &= c(p,q)(Contr(p,\Gamma_1,\Delta_1),_q Contr(p,\Gamma_2,\Delta_2)).
\end{aligned}$$

Fig. 2. Bunch Operations

Like in Fuzz, the !E rule propagates the scaling factor, but using the bunch structure. Rather than adding the two environments, we splice one into the other: the notation $\Gamma(\Delta)$ denotes a compound bunch where we plug in the bunch Δ into another bunch $\Gamma(\star)$ that has a single, distinguished hole $\star$. As we mentioned earlier, Bunched Fuzz has an explicit typing rule for contraction, whereas contraction in Fuzz is implicit in rules with multiple premises. Note also that we have unrestricted weakening. Finally, we have the rules for typing the return and bind primitives of the probabilistic types $\bigcirc_H$ and $\bigcirc_P$. Those for $\bigcirc_P$ are adapted from Fuzz, by using contraction instead of adding up the environments. The ones for $\bigcirc_H$ are similar, but use L^2 contraction instead, since that is the metric that enables composition for the Hellinger distance.

Let us now explain in which sense $\otimes_\infty$ corresponds to the & connective of Fuzz. We will need the following lemma:

Lemma 1 (Renaming). *Assume that there is a type derivation of $\Gamma \vdash e : \tau$ and that $\Gamma \approx \Gamma'$. Then there exists a derivation of $\Gamma' \vdash e[vars(\Gamma')/vars(\Gamma)] : \tau$.*

Now, the & connective in Fuzz supports two operations, projections and pairing. The connective $\otimes_\infty$ of Bunched Fuzz also supports these operations, but as derived forms. First, projections can be encoded by defining $\pi_i(e)$ for $i = 1,2$ as **let** $(x_1, x_2) = e$ **in** x_i. Second, for pairing assume we have two derivations of $\Gamma \vdash e_i : \sigma_i$ for $i = 1,2$, and let Γ' be an environment obtained from Γ by

$$\frac{s \geq 1}{[x : \tau]_s \vdash x : \tau}\ \textsc{Axiom} \qquad \frac{}{\cdot \vdash r : \mathbb{R}}\ \mathbb{R}\mathrm{I} \qquad \frac{}{\cdot \vdash () : 1}\ 1\mathrm{I}$$

$$\frac{\Gamma\ ,_p [x : \tau]_1 \vdash e : \sigma}{\Gamma \vdash \lambda x.e : \tau \multimap_p \sigma}\ \multimap\mathrm{I} \qquad \frac{\Gamma \vdash f : \tau \multimap_p \sigma \qquad \Delta \vdash e : \tau}{\Gamma ,_p \Delta \vdash f\ e : \sigma}\ \multimap\mathrm{E}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Delta \vdash e_2 : \sigma}{\Gamma\ ,_p \Delta \vdash (e_1, e_2) : \tau \otimes_p \sigma}\ \otimes\mathrm{I} \qquad \frac{\Delta \vdash e_1 : \tau \otimes_p \sigma \qquad \Gamma([x : \tau]_s\ ,_p [y : \sigma]_s) \vdash e_2 : \rho}{\Gamma(s\Delta) \vdash \mathbf{let}\ (x, y) = e_1\ \mathbf{in}\ e_2 : \rho}\ \otimes\mathrm{E}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{inj}_1 e : \tau \oplus \sigma}\ \oplus_1\mathrm{I} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathbf{inj}_2 e : \tau \oplus \sigma}\ \oplus_2\mathrm{I}$$

$$\frac{\Gamma \vdash e_1 : \tau \oplus \sigma \qquad \Delta([x : \tau]_s) \vdash e_2 : \rho \qquad \Delta([y : \sigma]_s) \vdash e_3 : \rho}{\Delta(s\Gamma) \vdash \mathbf{case}\ e_1\ \mathbf{of}\ x.\ e_2 \mid y.\ e_3 : \rho}\ \oplus\mathrm{E}$$

$$\frac{\Gamma \vdash e : \tau}{s\Gamma \vdash !e : !_s\tau}\ !\mathrm{I} \qquad \frac{\Gamma \vdash e_1 : !_r\tau \qquad \Delta([x : \tau]_{rs}) \vdash e_2 : \sigma}{\Delta(s\Gamma) \vdash \mathbf{let}\ !x = e_1\ \mathbf{in}\ e_2 : \sigma}\ !\mathrm{E}$$

$$\frac{\Gamma(\Delta\ ,_p \Delta') \vdash e : \tau \qquad \Delta \approx \Delta'}{\Gamma(Contr(p, \Delta, \Delta')) \vdash e[vars(\Delta')/vars(\Delta)] : \tau}\ \textsc{Contr} \qquad \frac{\Gamma(\cdot) \vdash e : \tau}{\Gamma(\Delta) \vdash e : \tau}\ \textsc{Weak}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \leftrightsquigarrow \Gamma'}{\Gamma' \vdash e : \tau}\ \textsc{Exch}$$

$$\frac{\begin{array}{c}\Gamma \approx \Delta \\ \Gamma \vdash e_1 : \bigcirc_P \tau \qquad \Delta ,_p [x : \tau]_s \vdash e_2 : \bigcirc_P \sigma\end{array}}{Contr(1, \Gamma, \Delta) \vdash \mathbf{mlet}\ x = e_1\ \mathbf{in}\ e_2 : \bigcirc_P \sigma}\ \textsc{Bind-P} \frac{\Gamma \vdash e : \tau}{\infty\Gamma \vdash \mathbf{return}\ e : \bigcirc_P \tau}\ \textsc{Return-P}$$

$$\frac{\begin{array}{c}\Gamma \approx \Delta \\ \Gamma \vdash e_1 : \bigcirc_H \tau \qquad \Delta ,_p [x : \tau]_s \vdash e_2 : \bigcirc_H \sigma\end{array}}{Contr(2, \Gamma, \Delta) \vdash \mathbf{mlet}\ x = e_1\ \mathbf{in}\ e_2 : \bigcirc_H \sigma}\ \textsc{Bind-H} \frac{\Gamma \vdash e : \tau}{\infty\Gamma \vdash \mathbf{return}\ e : \bigcirc_H \tau}\ \textsc{Return-H}$$

Fig. 3. Bunched Fuzz typing rules

renaming all variables to fresh ones. Then we have $\Gamma \approx \Gamma'$ and thus

$$\dfrac{\Gamma \vdash e_1 : \sigma_1 \qquad \dfrac{\Gamma \vdash e_2 : \sigma_2 \qquad \Gamma \approx \Gamma'}{\Gamma' \vdash e_2[vars(\Gamma')/vars(\Gamma)] : \sigma_2}\ \textsc{Lemma } 1}{\dfrac{\Gamma ,_\infty \Gamma' \vdash (e_1, e_2[vars(\Gamma')/vars(\Gamma)]) : \sigma_1 \otimes_\infty \sigma_2}{Contr(\infty, \Gamma, \Gamma') \vdash (e_1, e_2) : \sigma_1 \otimes_\infty \sigma_2}\ \textsc{Contr}}\ \otimes\mathrm{I}$$

Note that we have defined $\sqrt[\infty]{x^\infty + y^\infty} = \max(x, y)$ by taking the limit of $\sqrt[p]{x^p + y^p}$ when p goes to infinity, and thus we have $Contr(\infty, \Gamma, \Gamma') = \Gamma$. Therefore the pairing rule of & is derivable for $\otimes_\infty$.

4 Semantics

Having defined the syntax of Bunched Fuzz and its type system, we are ready to present its semantics. We opt for a denotational formulation, where types τ and bunches Γ are interpreted as metric spaces $[\![\tau]\!]$ and $[\![\Gamma]\!]$, and a derivation π of $\Gamma \vdash e : \tau$ is interpreted as a non-expansive function $[\![\pi]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$. For space reasons, we do not provide an operational semantics for the language, but we foresee no major difficulties in doing so, since the term language is mostly inherited from Fuzz, which does have a denotational semantics proved sound with respect to an operational semantics [4].

Types Each type τ is interpreted as a metric space $[\![\tau]\!]$ in a compositional fashion, by mapping each type constructor to the corresponding operation on metric spaces defined in Figure 4. We now explain these definitions.

The operations of the first four lines of Figure 4 come from prior work on Fuzz [4,3]. The definition of $\otimes_p$ uses as carrier set the cartesian product, just as $\otimes$ in previous works, but endows it with the L^p distance, defined in Section 2.4. In the particular case of $p = 1$, $\otimes_1$ is the same as $\otimes$.

As for $\multimap_p$, we want to define it in such a way that currying and uncurrying work with respect to $\otimes_p$, which will allow us to justify the introduction and elimination forms for that connective. For that we first choose as carrier set the set $A \multimap B$ of non-expansive functions from A to B. This set carries the metric

$$\begin{aligned} &d_{A \multimap_p B}(f, g) \\ &= \inf\{r \in \mathbb{R}^{\geq 0}_{\infty} \mid \forall x, y \in A, d_B(f(x), g(y)) \leq \sqrt[p]{r^p + d_A(x, y)^p}\} \end{aligned} \tag{11}$$

This metric is dictated by the type of the application operator in the L^p norm: $(A \multimap_p B) \otimes_p A \multimap B$. Intuitively, if f and g are at distance r, and we want application to be non-expansive, we need to satisfy $d_B(f(x), g(y)) \leq \sqrt[p]{r^p + d_A(x, y)^p}$ for every $x, y \in A$. The above definition says that we pick the distance to be the smallest possible r that makes this work. Note that this choice is forced upon us: in category-theoretic jargon, the operations of currying and uncurrying, which are intimately tied to the application operator, correspond to an adjunction between two functors, which implies that any other metric space that yields a similar adjunction with respect to $\otimes_p$ must be isomorphic to $\multimap_p$. In particular, this implies that its metric will be the same as the one of $\multimap_p$.

For $\bigcirc_P A$ and $\bigcirc_H A$ the carrier set is the set DA of discrete distributions over A. As to the metric on the carrier set, the interpretation of $\bigcirc_P$ uses the max divergence, used in the definition of differential privacy (see Sect. 2.2). The interpretation of $\bigcirc_H$ uses instead the Hellinger distance (see e.g. [3]):

$$\mathsf{HD}_A(\mu,\nu) \triangleq \sqrt{\frac{1}{2}\sum_{x\in A} |\sqrt{\mu(x)} - \sqrt{\nu(x)}|^2} \tag{12}$$

Space X	$\lvert X\rvert$	$d_X(x,y)$
1	$\{*\}$	0
$\mathbb{R}$	$\mathbb{R}$	$\lvert x-y\rvert$
$!_s A$	$\lvert A\rvert$	$\begin{cases} s\cdot d_A(x,y) \text{ if } s\neq\infty \\ \infty \text{ if } s=\infty, x\neq y\in A \\ 0 \text{ if } s=\infty, x=y\in A\end{cases}$
$A\oplus B$	$\lvert A\rvert+\lvert B\rvert$	$\begin{cases} d_A(x,y) \text{ if } x,y\in A \\ d_B(x,y) \text{ if } x,y\in B \\ \text{else } \infty\end{cases}$
$A\otimes_p B$	$\lvert A\rvert\times\lvert B\rvert$	$\sqrt[p]{d_A(\pi_1(x),\pi_1(y))^p + d_B(\pi_2(x),\pi_2(y))^p}$
$A\multimap_p B$	$A\multimap B$	cf. Equation (11)
$\bigcirc_P A$	DA	$\mathsf{MD}_A(x,y)$; cf. Equation (2)
$\bigcirc_H A$	DA	$\mathsf{HD}_A(x,y)$; cf. Equation (12)

Fig. 4. Operations on metric spaces for interpreting types

Bunches The interpretation of bunches is similar to that of types. Variables correspond to scaled metric spaces, whereas $,_p$ corresponds to $\otimes_p$:

$$[\![\cdot]\!] = 1 \qquad [\![[x:\tau]_s]\!] = !_s[\![\tau]\!] \qquad [\![\Gamma_1 ,_p \Gamma_2]\!] = [\![\Gamma_1]\!] \otimes_p [\![\Gamma_2]\!].$$

One complication compared to prior designs is the use of an explicit exchange rule, which is required to handle the richer structure of contexts. Semantically, each use of exchange induces an isomorphism of metric spaces:

Theorem 3. *Each derivation of $\Gamma \leftrightsquigarrow \Delta$ corresponds to an isomorphism of metric spaces $[\![\Gamma]\!] \cong [\![\Delta]\!]$.*

Before stating the interpretation of typing derivations, we give an overview of important properties of the above constructions that will help us prove the soundness of the interpretation.

Scaling Much like in prior work [4,3], we can check the following equations:

Proposition 4.

$$!_{s_1}!_{s_2}A = \,!_{s_1 \cdot s_2}A \qquad !_s(A \oplus B) = !_sA \oplus !_sB \qquad !_s(A \otimes_p B) = !_sA \otimes_p !_sB.$$

Moreover, an s-sensitive function from A to B is the same thing as a non-expansive function of type $!_sA \multimap B$.

Proposition 5. *For every bunch Γ, we have $[\![s\Gamma]\!] = !_s[\![\Gamma]\!]$.*

Tensors The properties on L^p distances allow us to relate product types with different values of p.

Proposition 6. *[Subtyping of tensors]*

1. *Let A, B be two metric spaces and $p, q \in \mathbb{R}^{\geq 1}_{\infty}$ with $p \leq q$. Then the identity map on pairs belongs to the two following spaces:*

$$A \otimes_p B \multimap A \otimes_q B \qquad\qquad !_{2^{1/p-1/q}}(A \otimes_q B) \multimap A \otimes_p B.$$

2. *In particular, when $p = 1$ and $q = 2$, the identity map belongs to:*

$$A \otimes_1 B \multimap A \otimes_2 B \qquad\qquad !_{\sqrt{2}}(A \otimes_2 B) \multimap A \otimes_1 B.$$

Proof. For (1), the fact that the identity belongs to the first space follows from the fact that $d_q(x, y) \leq d_p(x, y)$, by Proposition 3 (Equation (9)). The second claim is derived from Proposition 3 (Equation (9)) in the case $n = 2$.

Remark 1. Proposition 6 allows us to relate different spaces of functions with multiple arguments. For example,

$$(A \otimes_2 B \multimap C) \subseteq (A \otimes_1 B \multimap C) \quad (A \otimes_1 B \multimap C) \subseteq (!_{\sqrt{2}}(A \otimes_2 B) \multimap C).$$

Bunched Fuzz does not currently exploit these inclusions in any significant way, but we could envision extending the system with a notion of subtyping to further simplify the use of multiple product metrics in a single program.

We also have the following result, which is instrumental to prove the soundness of the contraction rule.

Proposition 7. *Let X, Y, Z, W be metric spaces, and $p, q \in \mathbb{R}^{\geq 1}_{\infty}$ with $p \neq \infty$. The canonical isomorphism of sets $(X \times Y) \times (Z \times W) \cong (X \times Z) \times (Y \times W)$, which swaps the second and third components, is a non-expansive function of type $!_{c(p,q)}((X \otimes_q Y) \otimes_p (Z \otimes_q W)) \to (X \otimes_p Z) \otimes_q (Y \otimes_p W)$, where $c(p, q)$ is defined as in Figure 2.*

Proof. First, suppose that $p \leq q$. Then we can write the isomorphism as a composite of the following non-expansive functions:

$$\begin{aligned}
&!_{c(p,q)}((X \otimes_q Y) \otimes_p (Z \otimes_q W) \\
&\to !_{c(p,q)}((X \otimes_q Y) \otimes_q (Z \otimes_q W)) && \text{Proposition 6} \\
&\cong !_{c(p,q)}((X \otimes_q Z) \otimes_q (Y \otimes_q W)) && \text{assoc., comm. of } \otimes_q \\
&= !_{c(p,q)}(X \otimes_q Z) \otimes_q !_{c(p,q)}(Y \otimes_q W) && \text{Proposition 4} \\
&= (X \otimes_p Z) \otimes_q (Y \otimes_p W) && \text{Proposition 6.}
\end{aligned}$$

Otherwise, $p > q$, and we reason as follows.

$$\begin{aligned}
&!_{c(p,q)}((X \otimes_q Y) \otimes_p (Z \otimes_q W) \\
&\to !_{c(p,q)}((X \otimes_p Y) \otimes_q (Z \otimes_p W)) && \text{Proposition 6} \\
&\cong !_{c(p,q)}((X \otimes_p Z) \otimes_p (Y \otimes_p W)) && \text{assoc., comm. of } \otimes_p \\
&= (X \otimes_p Z) \otimes_q (Y \otimes_p W) && \text{Proposition 6.}
\end{aligned}$$

One can then prove the following property:

Proposition 8. *Suppose that we have two bunches $\Gamma \approx \Delta$. The carrier sets of $[\![\Gamma]\!]$ and $[\![\Delta]\!]$ are the same. Moreover, for any p, the diagonal function $\delta(x) = (x, x)$ is a non-expansive function of type $[\![Contr(p, \Gamma, \Delta)]\!] \to [\![\Gamma]\!] \otimes_p [\![\Delta]\!]$.*

Function Types The metric on $\multimap_p$ can be justified by the following result:

Proposition 9. *For every metric space X and every $p \in \mathbb{R}^{\geq 1}_{\infty}$, there is an adjunction of type $(-) \otimes_p X \dashv X \multimap_p (-)$ in Met given by currying and uncurrying. (Both constructions on metric spaces are extended to endofunctors on Met in the obvious way.)*

Because right adjoints are unique up to isomorphism, this definition is a direct generalization of the metric on functions used in Fuzz [23,4,3], which corresponds to $\multimap_1$.

Theorem 4. *Suppose that A and B are* proper *metric spaces, and let $f, g : A \to B$ be non-expansive. Then $d_{A \multimap_1 B}(f, g) = \sup_x d_B(f(x), g(x))$.*

We conclude with another subtyping result involving function spaces.

Theorem 5. *For all non-expansive functions $f, g \in A \to B$ and $p \geq 1$, we have $d_{A \multimap_1 B}(f, g) \leq d_{A \multimap_p B}(f, g)$. In particular, the identity function is a non-expansive function of type $(A \multimap_p B) \to (A \multimap_1 B)$.*

Probability Distributions Prior work [3] proves that the return and bind operations on probability distributions can be seen as non-expansive functions:

$$\eta : !_\infty A \to \bigcirc_P A$$
$$(-)^\dagger(-) : (!_\infty A \multimap_1 \bigcirc_P B) \otimes_1 \bigcirc_P A \to \bigcirc_P B.$$

These properties ensure the soundness of the typing rules for $\bigcirc_P$ in Fuzz, and also in Bunched Fuzz. For $\bigcirc_H$, we can use the following composition principle.

Theorem 6. *The following types are sound for the monadic operations on distributions, seen as non-expansive operations, for any $p \geq 1$:*

$$\eta : !_\infty A \to \bigcirc_H A$$
$$(-)^\dagger(-) : (!_\infty A \multimap_p \bigcirc_H B) \otimes_2 \bigcirc_H A \to \bigcirc_H B.$$

Derivations Finally, a derivation tree builds a function from the context's space to the subject's space. In the following definition, we use the metavariables γ and δ to denote variable assignments—that is, mappings from the variables of environments Γ and Δ to elements of the corresponding metric spaces. We use $\gamma(\delta)$ to represent an assignment in $[\![\Gamma(\Delta)]\!]$ that is decomposed into two assignments $\gamma(\star)$ and δ corresponding to the $\Gamma(\star)$ and Δ portions. Finally, we use the λ-calculus notation $f\ x$ to denote a function f being applied to the value x.

Definition 1. *Given a derivation π proving $\Gamma \vdash e : \tau$, its interpretation $[\![\pi]\!] \in [\![\Gamma]\!] \to [\![\tau]\!]$ is given by structural induction on π as follows:*

$$\begin{array}{ll}
[\![Axiom]\!] \triangleq \lambda x.\ x & [\![\mathbb{R}I]\!] \triangleq \lambda().\ r \in \mathbb{R} \\
[\![\multimap I\ \pi]\!] \triangleq \lambda\gamma.\ \lambda x.\ [\![\pi]\!]\ (\gamma, x) & [\![\multimap E\ \pi_1\ \pi_2]\!] \triangleq \lambda(\gamma,\delta).\ [\![\pi_2]\!]\ \gamma\ ([\![\pi_1]\!]\ \delta) \\
[\![1I]\!] \triangleq \lambda().\ () & [\![\otimes I\ \pi_1\ \pi_2]\!] \triangleq \lambda(\gamma,\delta).\ ([\![\pi_1]\!]\ \gamma), ([\![\pi_2]\!]\ \delta) \\
 & [\![\otimes E\ \pi_1\ \pi_2]\!] \triangleq \lambda\gamma(\delta).\ [\![\pi_2]\!]\ \gamma([\![\pi_1]\!]\delta) \\
[\![\oplus_i I\ \pi]\!] \triangleq \lambda\gamma.\ \boldsymbol{inj}_i[\![\pi]\!]\ \gamma & [\![\oplus E\ \pi_1\ \pi_2]\!] \triangleq \lambda\delta(\gamma).\ [[\![\pi_2]\!], [\![\pi_3]\!]](\delta([\![\pi_1]\!]\gamma)) \\
[\![!I\ \pi]\!] \triangleq [\![\pi]\!] & [\![!E\ \pi_1\ \pi_2]\!] \triangleq \lambda\ \delta(\gamma).\ [\![\pi_2]\!]\ \delta([\![\pi_1]\!]\ \gamma) \\
[\![Contr\ \pi]\!] \triangleq \lambda\gamma(\delta).\ [\![\pi]\!]\ \gamma(\delta,\delta) & [\![Weak\ \pi]\!] \triangleq \lambda\gamma(\delta).\ [\![\pi]\!]\ \gamma(\ ()\ ) \\
[\![Exch\ \pi]\!] \triangleq \lambda\gamma'.[\![\pi]\!]\phi_{\gamma'/\gamma}(\gamma') & [\![Bind\text{-}P\ \pi_1\ \pi_2]\!] \triangleq \lambda\gamma'.\ ([\![\pi_2]\!]\gamma')^\dagger([\![\pi_1]\!]\gamma') \\
[\![Return\text{-}P\ \pi]\!] \triangleq \lambda\gamma.\ \eta([\![\pi]\!]\ \gamma) &
\end{array}$$

where in $[\![Exch\ \pi]\!]$, the map $\phi_{\Gamma'/\Gamma}$ is the isomorphism defined by Theorem 3. and for the two last cases see definitions in equations (3) and (4) (Bind-H and Return-H are defined in the same way).

Theorem 7 (Soundness). *Given a derivation π proving $\Gamma \vdash e : \tau$, then $[\![\pi]\!]$ is a non-expansive function from the space $[\![\Gamma]\!]$ to the space $[\![\tau]\!]$.*

5 Examples

We now look at examples of programs that illustrate the use of L^p metrics.

Currying and Uncurrying Let us illustrate the use of higher-order functions with combinators for currying and uncurrying.

$$\begin{aligned}
curry &: ((\tau \otimes_p \sigma) \multimap_p \rho) \multimap (\tau \multimap_p \sigma \multimap_p \rho) \\
curry\ f\ x\ y &= f(x, y) \\
uncurry &: (\tau \multimap_p \sigma \multimap_p \rho) \multimap ((\tau \otimes_p \sigma) \multimap_p \rho). \\
uncurry\ f\ z &= \textbf{let}\ (x, y) = z\ \textbf{in}\ f\ x\ y
\end{aligned}$$

Note that the indices on $\otimes$ and $\multimap$ need to be the same. The reason can be traced back to the $\multimap$ E rule (cf. Figure 3), which uses the $,_p$ connective to eliminate $\multimap_p$ (cf. the currying and uncurrying derivation in the appendix of the full paper for a detailed derivation). If the indices do not agree, currying is not possible; in other words, we cannot in general soundly curry a function of type $\tau \otimes_p \sigma \multimap_q \rho$ to obtain something of type $\tau \multimap_p \sigma \multimap_q \rho$. However, if $q \leq p$, note that it would be possible to soundly view $\tau \otimes_q \sigma$ as a subtype of $\tau \otimes_p \sigma$, thanks to Proposition 6. In this case, we could then convert from $\tau \otimes_p \sigma \multimap_q \rho$ to $\tau \otimes_q \sigma \multimap_q \rho$ (note the variance), and then curry to obtain a function of type $\tau \multimap_q \sigma \multimap_q \rho$.

Precise sensitivity for functions with multiple arguments Another useful feature of Bunched Fuzz is that its contraction rule allows us to split sensitivities more accurately than if we used the contraction rule that is derivable in the original Fuzz. Concretely, suppose that we have a program $\lambda p.\mathbf{let}\ (x, y) = p\ \mathbf{in}\ f(x, y) + g(x, y)$, where f and g have types $f : (!_2\mathbb{R}) \otimes_2 \mathbb{R} \multimap \mathbb{R}$ and $g : \mathbb{R} \otimes_2 (!_2\mathbb{R}) \multimap \mathbb{R}$, and where we have elided the wrapping and unwrapping of ! types, for simplicity.

Let us sketch how this program is typed in Bunched Fuzz. Addition belongs to $\mathbb{R} \otimes_1 \mathbb{R} \multimap \mathbb{R}$, so by Proposition 6 it can also be given the type $!_{\sqrt{2}}(\mathbb{R} \otimes_2 \mathbb{R}) \multimap \mathbb{R}$. Thus, we can build the following derivation for the body of the program:

$$\textsc{Contr}\ \frac{\Gamma \vdash f(x_1, y_1) + g(x_2, y_2) : \mathbb{R}}{[x : \mathbb{R}]_{\sqrt{10}} ,_2 [y : \mathbb{R}]_{\sqrt{10}} \vdash f(x, y) + g(x, y) : \mathbb{R}}$$

where $\Gamma = ([x_1 : \mathbb{R}]_{2\sqrt{2}},_2 [y_1 : \mathbb{R}]_{\sqrt{2}}),_2 ([x_2 : \mathbb{R}]_{\sqrt{2}},_2 [y_2 : \mathbb{R}]_{2\sqrt{2}})$, and where we used contraction twice to merge the xs and ys. Note that $||(2\sqrt{2}, \sqrt{2})||_2 = \sqrt{8+2} = \sqrt{10}$, which is why the final sensitivities have this form. By contrast, consider how we might attempt to type this program directly in the original Fuzz. Let us assume that we are working in an extension of Fuzz with types for expressing the domains of f and g, similarly to the L^2 vector types of Duet [20]. Moreover, let us assume that we have coercion functions that allow us to cast from $(!_2\mathbb{R}) \otimes_2 (!_2\mathbb{R})$ to $(!_2\mathbb{R}) \otimes_2 \mathbb{R}$ and $\mathbb{R} \otimes_2 (!_2\mathbb{R})$. If we have a pair $p :!_2((!_2\mathbb{R}) \otimes_2 (!_2\mathbb{R}))$, we can split its sensitivity to call f and g and then combine their results with addition. However, this type is equivalent to $!_4(\mathbb{R} \otimes_2 \mathbb{R})$, which means that the program was given a worse sensitivity (since $\sqrt{10} < 4$). Of course, it would also have been possible to extend Fuzz with a series of primitives that implement precisely the management of sensitivities performed by bunches. However, here this low-level reasoning is handled directly by the type system.

Programming with matrices The Duet language [20] provides several matrix types with the L^1, L^2, or L^∞ metrics, along with primitive functions for manipulating them. In Bunched Fuzz, these types can be defined directly as follows: $\mathbb{M}_p[m, n] = \otimes_1^m \otimes_p^n \mathbb{R}$. Following Duet, we use the L^1 distance to combine the rows and the L^p distance to combine the columns. One advantage of having types for matrices defined in terms of more basic constructs is that we can program functions for manipulating them directly, without resorting to separate primitives. For example, we can define the following terms in the language:

$$\begin{aligned} \mathit{addrow} &: \mathbb{M}_p[1,n] \otimes_1 \mathbb{M}_p[m,n] \multimap \mathbb{M}_p[m+1,n] \\ \mathit{addcolumn} &: \mathbb{M}_1[1,m] \otimes_1 \mathbb{M}_1[m,n] \multimap \mathbb{M}_1[m,n+1] \\ \mathit{addition} &: \mathbb{M}_1[m,n] \otimes_1 \mathbb{M}_1[m,n] \multimap \mathbb{M}_1[m,n]. \end{aligned}$$

The first program, *addrow*, appends a vector, represented as a $1 \times n$ matrix, to the first row of a $m \times n$ matrix. The second program, *addcolumn*, is similar, but appends the vector as a column rather than a row. Because of that, it is restricted to L^1 matrices. Finally, the last program, *addition*, adds the elements of two matrices pointwise.

Vector addition over sets Let us now show an example of a Fuzz term for which using L^p metrics allows to obtain a finer sensitivity analysis. We consider sets of vectors in $\mathbb{R}^d$ and the function *vectorSum* which, given such a set, returns the vectorial sum of its elements. In Fuzz, this function can be defined via a summation primitive $\mathit{sum} : !_\infty(!_\infty\tau \multimap \mathbb{R}) \multimap \mathit{set}\,\tau \multimap \mathbb{R}$, which adds up the results of applying a function to each element of a set [23]. The definition is:

$$\begin{gathered} \mathit{vectorSum} : !_d\, \mathit{set}(\otimes_1^d \mathbb{R}) \multimap_1 \otimes_1^d \mathbb{R} \\ \mathit{vectorSum}\ s = (\mathit{sum}\ \pi_1\ s, \ldots, \mathit{sum}\ \pi_d\ s). \end{gathered}$$

Here, $\pi_i : \otimes_1^d \mathbb{R} \multimap \mathbb{R}$ denotes the i-th projection, which can be defined by destructing a product. Set types in Fuzz are equipped with the Hamming metric [23], where the distance between two sets is the number of elements by which they differ. Note that, to ensure that *sum* has bounded sensitivity, we need to clip the results of its function argument to the interval $[-1,1]$. Fuzz infers a sensitivity of d for this function because its argument is used with sensitivity 1 in each component of the tuple. In Bunched Fuzz, we can define the same function as above, but we also have the option of using a different L^p distance to define *vectorSum*, which leads to the type $!_{d^{1/p}}\, \mathit{set}(\otimes_p^d \mathbb{R}) \multimap \otimes_p^d \mathbb{R}$, with a sensitivity of $d^{1/p}$. For the sake of readability, we'll show how this term is typed in the case $d = 2$. By typing each term $(\mathit{sum}\ \pi_i\ z_i)$ and applying $(\otimes I)$ we get:

$$[z_1 : \mathit{set}(\mathbb{R} \otimes_p \mathbb{R})]_1 \,,_p [z_2 : \mathit{set}(\mathbb{R} \otimes_p \mathbb{R})]_1 \vdash (\mathit{sum}\ \pi_1\ z_1, \mathit{sum}\ \pi_2\ z_2) : \mathbb{R} \otimes_p \mathbb{R}.$$

By applying contraction we get: $[z : \mathit{set}(\mathbb{R} \otimes_p \mathbb{R})]_{2^{1/p}} \vdash (\mathit{sum}\ \pi_1\ z, \mathit{sum}\ \pi_2\ z) : \mathbb{R} \otimes_p \mathbb{R}$. The claimed type is finally obtained by $(!E)$ and $(\multimap I)$.

Computing distances Suppose that the type X denotes a proper metric space (that is, where the triangle inequality holds). Then we can incorporate its distance function in Bunched Fuzz with the type $X \otimes_1 X \multimap \mathbb{R}$. Indeed, let x, x', y and y' be arbitrary elements of X. Then

$$\begin{aligned} d_X(x,y) - d_X(x',y') &\le d_X(x,x') + d_X(x',y') + d_X(y',y) - d_X(x',y') \\ &= d_X(x,x') + d_X(y,y') = d_1((x,y),(x',y')). \end{aligned}$$

By symmetry, we also know that $d_X(x', y') - d_X(x, y) \leq d_1((x, y), (x', y'))$. Combined, these two facts show

$$d_{\mathbb{R}}(d_X(x, y), d_X(x', y')) = |d_X(x, y) - d_X(x', y')| \leq d_1((x, y), (x', y')),$$

which proves that d_X is indeed a non-expansive function.

Calibrating noise to L^p distance Hardt and Talwar [17] have proposed a generalization of the Laplace mechanism, called the K-norm mechanism, to create a differentially private variant of a database query $f : \mathtt{db} \to \mathbb{R}^d$. The difference is that the amount of noise added is calibrated to the sensitivity of f measured with the K norm, as opposed to the L^1 distance used in the original Laplace mechanism. When K corresponds to the L^p norm, we will call this the L^p-mechanism, following Awan and Slavkovich [1].

Definition 2. *Given $f : \mathtt{db} \to \mathbb{R}^d$ with L^p sensitivity s and $\epsilon > 0$, the L^p-mechanism is a mechanism that, given a database $D \in \mathtt{db}$, returns a probability distribution over $y \in \mathbb{R}^d$ with density given by:*

$$\frac{\exp(\frac{-\epsilon||f(D)-y||_p}{2s})}{\int \exp(\frac{-\epsilon||f(D)-y||_p}{2s})dy}$$

This mechanism returns with high probability (which depends on ϵ and on the sensitivity s) a vector $y \in \mathbb{R}^d$ which is close to $f(D)$ in L^p distance. Such a mechanism can be easily integrated in Bunched Fuzz through a primitive:

$$\mathtt{LpMech} : !_\infty(!_s\mathtt{dB} \multimap \otimes_p^d \mathbb{R}) \multimap !_\epsilon \mathtt{dB} \multimap \bigcirc_P(\otimes_p^d \mathbb{R})$$

(Strictly speaking, we would need some discretized version of the above distribution to incorporate the mechanism in Bunched Fuzz, but we'll ignore this issue in what follows.) The fact that `LpMech` satisfies ϵ-differential privacy follows from the fact that this mechanism is an instance of the *exponential mechanism* [18], a basic building block of differential privacy. It is based on a scoring function assigning a score to every pair consisting of a database and a potential output, and it attempts to return an output with approximately maximal score, given the input database. As shown by Gaboardi et al. [13], the exponential mechanism can be added as a primitive to Fuzz with type:

$$\mathtt{expmech} : !_\infty\, set(\mathcal{O}) \multimap !_\infty(!_\infty \mathcal{O} \multimap !_s\mathtt{dB} \multimap \mathbb{R}) \multimap !_\epsilon \mathtt{dB} \multimap \bigcirc_P \mathcal{O},$$

where $\mathcal{O}$ is the type of outputs. The function `LpMech` is an instance of the exponential mechanism where $\mathcal{O}$ is $\otimes_p^d \mathbb{R}$ and the score is $\lambda y \lambda D.||f(D) - y||_p$.

To define the L^p mechanism with this recipe, we need to reason about the sensitivity of this scoring function. In Fuzz, this would not be possible, since the language does not support reasoning about the sensitivity of f measured in the L^p distance. In Bunched Fuzz, however, this can be done easily. Below, we will see an example (Gradient descent) of how the L^p mechanism can lead to a finer privacy guarantee.

Gradient descent Let us now give an example where we use the L^p mechanism. An example of differentially private gradient descent example with linear model in Fuzz was given in [25] (see Sect. 4.1, 4.2 and Fig. 6 p. 16, Fig. 8 p.19). This algorithm proceeds by iteration. Actually it was given for an extended language called Adaptative Fuzz, but the code already gives an algorithm in (plain) Fuzz. We refer the reader to this reference for the description of all functions, and here we will only describe how one can adapt the algorithm to Bunched Fuzz.

Given a set of n records $x_i \in \mathbb{R}^d$, each with a *label* $y_i \in \mathbb{R}$, the goal is to find a parameter vector $\theta \in \mathbb{R}^d$ that minimizes the difference between the labels and their *estimates*, where the estimate of a label y_i is the inner product $\langle x_i, \theta \rangle$. That is, the goal is to minimize the loss function $L(\theta, (x, y)) = \frac{1}{n} \cdot \Sigma_{i=1}^{n} (\langle x_i, \theta \rangle - y_i)^2$. The algorithm starts with an initial parameter vector $(0, \ldots, 0)$ and it iteratively produces successive θ vectors until a termination condition is reached.

The Fuzz program uses the data-type *bag* τ representing bags or multisets over τ. A *bagmap* primitive is given for it. The type I is the unit interval $[0, 1]$. The main function is called *updateParameter* and updates one component of the model θ; it is computed in the following way:

– with the function *calcGrad* : $\mathtt{db} \to \mathbb{R}$, compute a component $(\nabla L(\theta, (x, y)))_j$ of the $\mathbb{R}^d$ vector $\nabla L(\theta, (x, y))$ [9].
– then Laplacian noise is postcomposed with *calcGrad* in the *updateParameter* function. This uses a privacy budget of 2ϵ. It has to be done for each one of the d components of $\nabla L(\theta, (x, y))$, thus on the whole, for one step, a privacy budget of $2d\epsilon$.
– The iterative procedure of gradient descent is given by the function *gradient* in Fig. 8 p. 19 of [25]. We forget here about the adaptative aspect and just consider iteration with a given number n of steps. In this case by applying n times *updateParameter* one gets a privacy budget of $2dn\epsilon$.

We modify the program as follows to check it in Bunched Fuzz and use the L^p-mechanism. Instead of computing over $\mathbb{R}$ we want to compute over $\otimes_p^d \mathbb{R}$ for a given $p \geq 1$, so $\mathbb{R}^d$ equipped with L^p distance. The records x_i are in $\otimes_p^d I$ and the labels y_i in I. The database type is $\mathtt{dB} = bag\ (I \otimes_p (\otimes_p^d I))$. The distance between two bags in $\mathtt{dB}$ is the number of elements by which they differ.

We assume a primitive *bagVectorSum* with type $!_{d^{1/p}} bag\ (\otimes_p^d I) \multimap \otimes_p^d \mathbb{R}$ (it could be defined as the *vectorSum* defined above for sets, using a *sum* primitive for bags). Given a bag m, (*bagVectorSum* m) returns the vectorial sum of all elements of m. We can check that the sensitivity of *bagVectorSum* is indeed $d^{1/p}$ because given two bags m and m' that are at distance 1, if we denote by u the vector by which they differ, we have:

$$d_{(\otimes_p^d \mathbb{R})}(bagVectorSum(m), bagVectorSum(m')) = ||u||_p \quad \leq (\Sigma_{j=1}^{d} 1)^{1/p} = d^{1/p}$$

By adapting the *calcGrad* Fuzz term of [25] using *bagVectorSum* we obtain a term *VectcalcGrad* with the Bunched Fuzz type $!_\infty \otimes_p^d \mathbb{R} \multimap !_{d^{1/p}} \mathtt{db} \multimap \otimes_p^d \mathbb{R}$.

[9] Actually *calcGrad* computes $(\nabla L(\theta, (x, y)))_j$ up to a multiplicative constant, 2/n, which is mutliplied afterwards in the *updateParameter* function.

Given a vector θ and a database (y, x), $VectcalcGrad$ computes the updated vector θ'. Finally we define the term $updateVector$ by adding noise to $VectcalcGrad$ using the the L^p-mechanism. Recall the type of `LpMech`: $!_\infty(!_s\texttt{db} \multimap \otimes_p^d\mathbb{R}) \multimap !_\epsilon\texttt{db} \multimap \bigcirc_P(\otimes_p^d\mathbb{R})$. We define $updateVector$ and obtain its type as follows:

$$updateVector = \lambda\theta.(\texttt{LpMech}\ (VectcalcGrad\ \theta)) : !_\infty \otimes_p^d \mathbb{R} \multimap !_\epsilon\texttt{db} \multimap \bigcirc_P(\otimes_p^d\mathbb{R})$$

By iterating $updateVector$ n times one obtains a privacy budget of $n\epsilon$.

6 Implementation

To experiment with the Bunched Fuzz design, we implemented a prototype for a fragment of the system based on DFuzz [13,2].[10] The type-checker generates a set of numeric constraints that serve as verification conditions to guarantee a valid typing. The implementation required adapting some of the current rules to an algorithmic formulation (found in the full version). In addition to the modifications introduced in the DFuzz type checker compared to its original version [13,2], we also made the following changes and simplifications:

- We did not include explicit contraction and weakening rules. Instead, the rules are combined with those for checking other syntactic constructs. To do away with an explicit contraction rule, in rules that have multiple antecedents, such as the $\otimes$I rule, we used the $Contr$ operator to combine the antecedents' environments, rather than using the p-concatenation operator for bunches.
- We did not include the rules for checking probabilistic programs with the Hellinger distance.
- Bound variables are always added at the top of the current environment, as in the $\multimap$I rule of the original rules; it is not possible to introduce new variables arbitrarily deep in the environment.

While, strictly speaking, the resulting system is incomplete with respect to the rules presented here, it is powerful enough to check an implementation of K-means that generalizes a previous version implemented for Fuzz [23]. On the other hand, because our implementation is based on the one of DFuzz, which features dependent types, we allow functions that are polymorphic on types, sizes and p parameters, which allows us to infer sensitivity information that depends on run-time sizes.

7 Related Work

Bunched Fuzz is inspired by BI, the logic of bunched implications [22], which has two connectives for combining contexts. Categorically, one of these connectives corresponds to a Cartesian product, whereas the other corresponds to a

[10] https://github.com/junewunder/bunched-fuzz

monoidal, or tensor product. While related to linear logic, the presence of the two context connectives allows BI to derive some properties that are not valid in linear logic. For example, the cartesian product does not distribute over sums in linear logic but it does distribute over sums in BI.

We have shown how the rules for such type systems are reminiscent of the ones used in type systems for the calcuclus of bunched implications [21], and for reasoning about categorical grammars [19]. Specifically, O'Hearn introduces a type system with two products and two arrows [21]. Typing environments are bunches of variable assignments with two constructors, corresponding to the two products. Our work can be seen as a generalization of O'Hearn's work to handle multiple products and to reason about program sensitivity.

Moot and Retoré [19, Chapter 5] introduce the multimodal Lambek calculus, which extends the non-associative Lambek calculus, a classical tool for describing categorical grammars. This generalization uses an indexed family of connectives and trees to represent environments. The main differences with our work are: our indexed products are associative and commutative, while theirs are not; our type system is affine; our type system includes a monad for probabilities which does not have a correspondent construction in their logic; our type system also possesses the graded comonad $!_s$ corresponding to the ! modality of linear logic, the interaction between this comonad and the bunches is non-trivial and it requires us to explicitly define a notion of contraction. Besides the fact that the main properties we study, metric interpretation and program sensitivity, are very different from the ones studied by the above authors, there are some striking similarities between the two systems.

A recent work by Bao et al. [5] introduced a novel bunched logic with indexed products and magic wands with a preorder between the indices. This logic is used as the assertion logic of a separation logic introduced to reason about negative dependence between random variables. The connectives studied in this work share some similarities with the ones we study here and it would be interesting to investigate further the similarities, especially from a model-theoretic perspective.

Because contexts in the original Fuzz type system are biased towards the L^1 distance, it is not obvious how Fuzz could express the composition principles of the Hellinger distance. Recent work showed how this could be amended via a *path construction* that recasts relational program properties as sensitivity properties [3]. Roughly speaking, instead of working directly with the Hellinger distance d_H, the authors consider a family of relations $R_\alpha = \{(\mu_1, \mu_2) \mid d_H(\mu_1, \mu_2) \leq \alpha\}$. Such a relation induces another metric on distributions, $d_{\alpha,H}$, where the distance between two distributions is the length of the shortest path connecting them in the graph corresponding to R_α. This allows them to express the composition principles of the Hellinger distance directly in the Fuzz type system, albeit at a cost: the type constructor for probability distributions is graded by the distance bound α. Thus, the sensitivity information of a randomized algorithm with respect to the Hellinger distance must also be encoded in the codomain of the function, as opposed to using just its domain, as done for the original privacy metric of Fuzz. By contrast, Bunched Fuzz does not require the grading α because it can express the composition principle of the Hellinger distance directly, thanks to the use of the L^2 distance on bunches.

Duet [20] can be seen as an extension of Fuzz to deal with more general privacy distances. It consists of a two-layer language: a sensitivity language and a privacy language. The sensitivity language is very similar to Fuzz. However, it also contains some basic primitives to manage vectors and matrices. As in Fuzz, the vector types come with multiple distances but differently from Fuzz, Duet also uses the L^2 distance. The main reason for this is that Duet also supports the Gaussian mechanism which calibrates the noise to the L^2 sensitivity of the function. Our work is inspired by this aspect of Duet, but it goes beyond it by giving a logical foundation to L^p vector distances. Another language inspired by Fuzz is the recently proposed Jazz [24]. Like Duet, this language has two products and primitives tailored to the L^2 sensitivity of functions for the Gaussian mechanism. Interestingly, this language uses contextual information to achieve more precise bounds on the sensitivities. The semantics of Jazz is different from the metric semantics we study here; however, it would be interesting to explore whether a similar contextual approach could be also used in a metric setting.

8 Conclusion and Future work

In this work we have introduced Bunched Fuzz, a type system for reasoning about program sensitivity in the style of Fuzz [23]. Bunched Fuzz extends the type theory of Fuzz by considering new type constructors for L^p distances and bunches to manage different products in typing environments. We have shown how this type system supports reasoning about both deterministic and probabilistic programs.

There are at least two directions that we would like to explore in future works. On the one hand, we would like to understand if the typing rules we introduced here could be of more general use in the setting of probabilistic programs. We have already discussed the usefulness for other directions in the deterministic case [19]. One way to approach this problem could be by looking at the family of products recently identified in [5]. These products give a model for a logic to reason about negative dependence between probabilistic variables. It would be interesting to see if the properties of these products match the one we have here.

On the other hand, we would like to understand if Bunched Fuzz can be used to reason about more general examples in differential privacy. One way to approach this problem could be to consider examples based on the use of Hellinger distance that have been studied in the literature on probabilistic inference [6].

Acknowledgements This material is based upon work supported by the NSF under Grant No. 1845803 and 2040249. The third author was partially supported by the french Program "Investissements d'avenir" (I-ULNE SITE / ANR-16-IDEX-0004 ULNE) managed by the National Research Agency.

References

1. Awan, J., Slavkovic, A.: Structure and sensitivity in differential privacy: Comparing k-norm mechanisms. Journal of the American Statistical Association (2020). https://doi.org/10.1080/01621459.2020.1773831, https://par.nsf.gov/biblio/10183971
2. Azevedo de Amorim, A., Gaboardi, M., Arias, E.J.G., Hsu, J.: Really natural linear indexed type checking. In: Tobin-Hochstadt, S. (ed.) Proceedings of the 26th 2014 International Symposium on Implementation and Application of Functional Languages, IFL '14, Boston, MA, USA, October 1-3, 2014. pp. 5:1–5:12. ACM (2014). https://doi.org/10.1145/2746325.2746335
3. Azevedo de Amorim, A., Gaboardi, M., Hsu, J., Katsumata, S.: Probabilistic relational reasoning via metrics. In: 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019. pp. 1–19. IEEE (2019). https://doi.org/10.1109/LICS.2019.8785715
4. Azevedo de Amorim, A., Gaboardi, M., Hsu, J., Katsumata, S., Cherigui, I.: A semantic account of metric preservation. In: POPL 2017. ACM (2017), http://dl.acm.org/citation.cfm?id=3009890
5. Bao, J., Gaboardi, M., Hsu, J., Tassarotti, J.: A separation logic for negative dependence. Proc. ACM Program. Lang. **6**(POPL) (jan 2022). https://doi.org/10.1145/3498719
6. Barthe, G., Farina, G.P., Gaboardi, M., Arias, E.J.G., Gordon, A., Hsu, J., Strub, P.: Differentially private bayesian programming. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 68–79. ACM (2016). https://doi.org/10.1145/2976749.2978371
7. Bousquet, O., Elisseeff, A.: Stability and generalization. J. Mach. Learn. Res. **2**, 499–526 (2002), http://jmlr.org/papers/v2/bousquet02a.html
8. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge University Press (March 2004)
9. Chaudhuri, S., Gulwani, S., Lublinerman, R., NavidPour, S.: Proving programs robust. In: Gyimóthy, T., Zeller, A. (eds.) SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011. pp. 102–112. ACM (2011). https://doi.org/10.1145/2025113.2025131
10. Csiszár, I., Shields, P.: Information theory and statistics: A tutorial. Foundations and Trends® in Communications and Information Theory **1**(4), 417–528 (2004). https://doi.org/10.1561/0100000004, http://dx.doi.org/10.1561/0100000004
11. Dwork, C., McSherry, F., Nissim, K., Smith, A.D.: Calibrating noise to sensitivity in private data analysis. In: Halevi, S., Rabin, T. (eds.) Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings. Lecture Notes in Computer Science, vol. 3876, pp. 265–284. Springer (2006). https://doi.org/10.1007/11681878_14
12. Dwork, C., Roth, A.: The algorithmic foundations of differential privacy. Found. Trends Theor. Comput. Sci. **9**(3-4), 211–407 (2014). https://doi.org/10.1561/0400000042

13. Gaboardi, M., Haeberlen, A., Hsu, J., Narayan, A., Pierce, B.C.: Linear dependent types for differential privacy. In: POPL '13. ACM (2013). https://doi.org/10.1145/2429069.2429113
14. Girard, J.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987). https://doi.org/10.1016/0304-3975(87)90045-4
15. Gonin, R., Money, A.H.: Nonlinear Lp-Norm Estimation. Marcel Dekker, Inc., USA (1989)
16. Haeberlen, A., Pierce, B.C., Narayan, A.: Differential privacy under fire. In: 20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings. USENIX Association (2011), http://static.usenix.org/events/sec11/tech/full_papers/Haeberlen.pdf
17. Hardt, M., Talwar, K.: On the geometry of differential privacy. In: Schulman, L.J. (ed.) Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010. pp. 705–714. ACM (2010). https://doi.org/10.1145/1806689.1806786
18. McSherry, F., Talwar, K.: Mechanism design via differential privacy. In: 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Providence, RI, USA, Proceedings. pp. 94–103. IEEE Computer Society (2007). https://doi.org/10.1109/FOCS.2007.41
19. Moot, R., Retoré, C.: The Logic of Categorial Grammars - A Deductive Account of Natural Language Syntax and Semantics, Lecture Notes in Computer Science, vol. 6850. Springer (2012). https://doi.org/10.1007/978-3-642-31555-8
20. Near, J.P., Darais, D., Abuah, C., Stevens, T., Gaddamadugu, P., Wang, L., Somani, N., Zhang, M., Sharma, N., Shan, A., Song, D.: Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360598
21. O'Hearn, P.W.: On bunched typing. J. Funct. Program. **13**(4), 747–796 (2003). https://doi.org/10.1017/S0956796802004495
22. O'Hearn, P.W., Pym, D.J.: The logic of bunched implications. Bull. Symb. Log. **5**(2) (1999). https://doi.org/10.2307/421090
23. Reed, J., Pierce, B.C.: Distance makes the types grow stronger: a calculus for differential privacy. In: ICFP 2010. ACM (2010). https://doi.org/10.1145/1863543.1863568
24. Toro, M., Darais, D., Abuah, C., Near, J., Olmedo, F., Tanter, É.: Contextual linear types for differential privacy. CoRR **abs/2010.11342** (2020), https://arxiv.org/abs/2010.11342
25. Winograd-Cort, D., Haeberlen, A., Roth, A., Pierce, B.C.: A framework for adaptive differential privacy. Proc. ACM Program. Lang. **1**(ICFP), 10:1–10:29 (2017). https://doi.org/10.1145/3110254
26. june wunder, Azevedo de Amorim, A., Baillot, P., Gaboardi, M.: Bunched fuzz: Sensitivity for vector metrics. CoRR **abs/2202.01901** (2022), https://arxiv.org/abs/2202.01901

Fast and Correct Gradient-Based Optimisation for Probabilistic Programming via Smoothing

Basim Khajwal[1], C.-H. Luke Ong[1,2], and Dominik Wagner[1](✉)

[1] University of Oxford, Oxford, UK
dominik.wagner@cs.ox.ac.uk
[2] NTU, Singapore, Singapore

Abstract. We study the foundations of variational inference, which frames posterior inference as an optimisation problem, for probabilistic programming. The dominant approach for optimisation in practice is stochastic gradient descent. In particular, a variant using the so-called reparameterisation gradient estimator exhibits fast convergence in a traditional statistics setting. Unfortunately, discontinuities, which are readily expressible in programming languages, can compromise the correctness of this approach. We consider a simple (higher-order, probabilistic) programming language with conditionals, and we endow our language with both a measurable and a *smoothed* (approximate) value semantics. We present type systems which establish technical pre-conditions. Thus we can prove stochastic gradient descent with the reparameterisation gradient estimator to be correct when applied to the smoothed problem. Besides, we can solve the original problem up to any error tolerance by choosing an accuracy coefficient suitably. Empirically we demonstrate that our approach has a similar convergence as a key competitor, but is simpler, faster, and attains orders of magnitude reduction in work-normalised variance.

Keywords: probabilistic programming · variational inference · reparameterisation gradient · value semantics · type systems.

1 Introduction

Probabilistic programming is a programming paradigm which has the vision to make statistical methods, in particular Bayesian inference, accessible to a wide audience. This is achieved by a separation of concerns: the domain experts wishing to gain statistical insights focus on modelling, whilst the inference is performed automatically. (In some recent systems [4,9] users can improve efficiency by writing their own inference code.)

In essence, probabilistic programming languages extend more traditional programming languages with constructs such as **score** or **observe** (as well as **sample**) to define the prior $p(\mathbf{z})$ and likelihood $p(\mathbf{x} \mid \mathbf{z})$. The task of inference is to derive the posterior $p(\mathbf{z} \mid \mathbf{x})$, which is in principle governed by Bayes' law yet usually intractable.

Whilst the paradigm was originally conceived in the context of statistics and Bayesian machine learning, probabilistic programming has in recent years

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 479–506, 2023.
https://doi.org/10.1007/978-3-031-30044-8_18

proven to be a very fruitful subject for the programming language community. Researchers have made significant theoretical contributions such as underpinning languages with rigorous (categorical) semantics [35,34,15,37,12,10] and investigating the correctness of inference algorithms [16,7,22]. The latter were mostly designed in the context of "traditional" statistics and features such as conditionals, which are ubiquitous in programming, pose a major challenge for correctness.

Inference algorithms broadly fall into two categories: Markov chain Monte Carlo (MCMC), which yields a sequence of samples asymptotically approaching the true posterior, and variational inference.

Variational Inference. In the variational inference approach to Bayesian statistics [40,30,5,6], the problem of approximating difficult-to-compute posterior probability distributions is transformed to an optimisation problem. The idea is to approximate the posterior probability $p(\mathbf{z} \mid \mathbf{x})$ using a family of "simpler" densities $q_{\boldsymbol{\theta}}(\mathbf{z})$ over the latent variables $\mathbf{z}$, parameterised by $\boldsymbol{\theta}$. The optimisation problem is then to find the parameter $\boldsymbol{\theta}^*$ such that $q_{\boldsymbol{\theta}^*}(\mathbf{z})$ is "closest" to the true posterior $p(\mathbf{z} \mid \mathbf{x})$. Since the variational family may not contain the true posterior, $q_{\boldsymbol{\theta}^*}$ is an approximation in general. In practice, variational inference has proven to yield good approximations much faster than MCMC.

Formally, the idea is captured by minimising the *KL-divergence* [30,5] between the variational approximation and the true posterior. This is equivalent to maximising the ELBO function, which only depends on the joint distribution $p(\mathbf{x}, \mathbf{z})$ and *not* the posterior, which we seek to infer after all:

$$\mathrm{ELBO}_{\boldsymbol{\theta}} := \mathbb{E}_{\mathbf{z}\sim q_{\boldsymbol{\theta}}(\mathbf{z})}[\log p(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\theta}}(\mathbf{z})] \tag{1}$$

Gradient Based Optimisation. In practice, variants of *Stochastic Gradient Descent (SGD)* are frequently employed to solve optimisation problems of the following form: $\mathrm{argmin}_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{s}\sim q(\mathbf{s})}[f(\boldsymbol{\theta}, \mathbf{s})]$. In its simplest version, SGD follows Monte Carlo estimates of the gradient in each step:

$$\boldsymbol{\theta}_{k+1} := \boldsymbol{\theta}_k - \gamma_k \cdot \underbrace{\frac{1}{N}\sum_{i=1}^{N} \nabla_\theta f\left(\boldsymbol{\theta}_k, \mathbf{s}_k^{(i)}\right)}_{\text{gradient estimator}}$$

where $\mathbf{s}_k^{(i)} \sim q\left(\mathbf{s}_k^{(i)}\right)$ and γ_k is the *step size*.

For the correctness of SGD it is crucial that the estimation of the gradient is *unbiased*, i.e. correct in expectation:

$$\mathbb{E}_{\mathbf{s}^{(1)},\ldots,\mathbf{s}^{(N)}\sim q}\left[\frac{1}{N}\sum_{i=1}^{N} \nabla_\theta f\left(\boldsymbol{\theta}, \mathbf{s}^{(i)}\right)\right] = \nabla_\theta \mathbb{E}_{\mathbf{s}\sim q(\mathbf{s})}[f(\boldsymbol{\theta}, \mathbf{s})]$$

This property, which is about commuting differentiation and integration, can be established by the dominated convergence theorem [21, Theorem 6.28].

Note that we cannot directly estimate the gradient of the ELBO in Eq. (1) with Monte Carlo because the distribution w.r.t. which the expectation is taken also depends on the parameters. However, the so-called *log-derivative trick* can be used to derive an unbiased estimate, which is known as the *Score* or *REINFORCE* estimator [31,38,27,28].

Reparameterisation Gradient. Whilst the score estimator has the virtue of being very widely applicable, it unfortunately suffers from high variance, which can cause SGD to yield very poor results[3].

The *reparameterisation gradient estimator*—the dominant approach in variational inference—reparameterises the latent variable $\mathbf{z}$ in terms of a base random variable $\mathbf{s}$ (viewed as the entropy source) via a diffeomorphic transformation $\boldsymbol{\phi}_{\boldsymbol{\theta}}$, such as a location-scale transformation or cumulative distribution function. For example, if the distribution of the latent variable z is a Gaussian $\mathcal{N}(z \mid \mu, \sigma^2)$ with parameters $\boldsymbol{\theta} = \{\mu, \sigma\}$ then the location-scale transformation using the standard normal as the base distribution gives rise to the reparameterisation

$$z \sim \mathcal{N}(z \mid \mu, \sigma^2) \iff z = \phi_{\mu,\sigma}(s), \quad s \sim \mathcal{N}(0,1). \tag{2}$$

where $\phi_{\mu,\sigma}(s) \coloneqq s \cdot \sigma + \mu$. The key advantage of this setup (often called "reparameterisation trick" [20,36,32]) is that we have removed the dependency on $\boldsymbol{\theta}$ from the distribution w.r.t. which the expectation is taken. Therefore, we can now differentiate (by backpropagation) with respect to the parameters $\boldsymbol{\theta}$ of the variational distributions using Monte Carlo simulation with draws from the base distribution $\mathbf{s}$. Thus, succinctly, we have

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{z}\sim q_{\boldsymbol{\theta}}(\mathbf{z})}[f(\boldsymbol{\theta}, \mathbf{z})] = \nabla_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{s}\sim q(\mathbf{s})}[f(\boldsymbol{\theta}, \boldsymbol{\phi}_{\boldsymbol{\theta}}(\mathbf{s}))] = \mathbb{E}_{\mathbf{s}\sim q(\mathbf{s})}[\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}, \boldsymbol{\phi}_{\boldsymbol{\theta}}(\mathbf{s}))]$$

The main benefit of the reparameterisation gradient estimator is that it has a significantly lower variance than the score estimator, resulting in faster convergence.

Bias of the Reparameterisation Gradient. Unfortunately, the reparameterisation gradient estimator is biased for non-differentiable models [23], which are readily expressible in programming languages with conditionals:

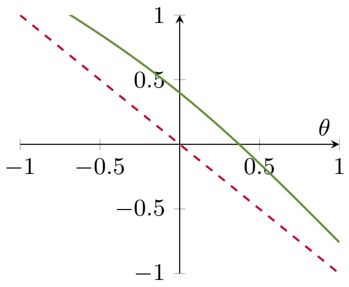
Example 1. The counterexample in [23, Proposition 2], where the objective function is the ELBO for a non-differentiable model, can be simplified to

$$f(\theta, s) = -0.5 \cdot \theta^2 + \begin{cases} 0 & \text{if } s + \theta < 0 \\ 1 & \text{otherwise} \end{cases}$$

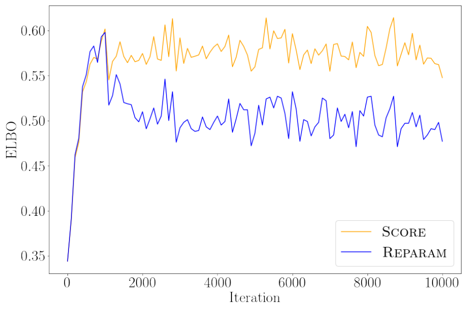
Observe that (see Fig. 1a):

$$\nabla_\theta \mathbb{E}_{s\sim\mathcal{N}(0,1)}\left[f(\theta, s)\right] = -\theta + \mathcal{N}(-\theta \mid 0, 1) \neq -\theta = \mathbb{E}_{s\sim\mathcal{N}(0,1)}\left[\nabla_\theta f(\theta, s)\right]$$

3 see e.g. Fig. 5a or [28]



(a) Dashed red: biased estimator $\mathbb{E}_{s\sim\mathcal{N}(0,1)}\left[\nabla_\theta f(\theta, s)\right]$, solid green: true gradient $\nabla_\theta\, \mathbb{E}_{s\sim\mathcal{N}(0,1)}\left[f(\theta, s)\right]$.



(b) ELBO trajectories (higher means better) obtained with our implementation (cf. Section 7)

Fig. 1: Bias of the reparameterisation gradient estimator for Example 1.

Crucially *this may compromise convergence to critical points or maximisers*: even if we can find a point where the gradient estimator vanishes, it may not be a critical point (let alone optimum) of the original optimisation problem (cf. Fig. 1b)

Informal Approach

As our starting point we take a variant of the simply typed lambda calculus with reals, conditionals and a sampling construct. We abstract the optimisation of the ELBO to the following generic optimisation problem

$$\operatorname{argmin}_{\boldsymbol{\theta}}\, \mathbb{E}_{\mathbf{s}\sim\mathcal{D}}[[\![M]\!](\boldsymbol{\theta}, \mathbf{s})] \tag{3}$$

where $[\![M]\!]$ is the value function [7,26] of a program M and $\mathcal{D}$ is independent of the parameters $\boldsymbol{\theta}$ and it is determined by the distributions from which M samples. Owing to the presence of conditionals, the function $[\![M]\!]$ may not be continuous, let alone differentiable.

Example 1 can be expressed as

$$(\lambda z.\, -0.5 \cdot \theta^2 + (\textbf{if}\; z < 0\; \textbf{then}\; 0\; \textbf{else}\; 1))\; (\textbf{sample}_{\mathcal{N}} + \theta)$$

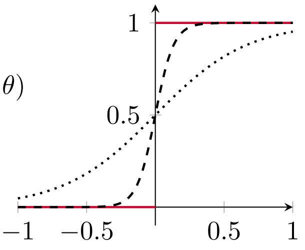


Fig. 2: (Logistic) sigmoid function σ_η (dotted: $\eta = \frac{1}{3}$, dashed: $\eta = \frac{1}{15}$) and the Heaviside step function (red, solid).

Our approach is based on a denotational semantics $[\![(-)]\!]_\eta$ (for accuracy coefficient $\eta > 0$) of programs in the (new) cartesian closed category **VectFr**, which generalises smooth manifolds and extends Frölicher spaces (see e.g. [13,33]) with a vector space structure.

Intuitively, we replace the Heaviside step-function usually arising in the interpretation of conditionals by smooth approximations. In particular, we interpret the conditional of Example 1 as

$$[\![\textbf{if}\; s + \theta < 0\; \textbf{then}\; \underline{0}\; \textbf{else}\; \underline{1}]\!]_\eta(\theta, s) := \sigma_\eta(s + \theta)$$

where σ_η is a smooth function. For instance we can choose $\sigma_\eta(x) := \sigma(\frac{x}{\eta})$ where $\sigma(x) := \frac{1}{1+\exp(-x)}$ is the (logistic) sigmoid function (cf. Fig. 2). Thus, the program M is interpreted by a smooth function $[\![M]\!]_\eta$, for which the reparameterisation gradient may be estimated unbiasedly. Therefore, we apply stochastic gradient descent on the smoothed program.

Contributions

The high-level contribution of this paper is laying a theoretical foundation for correct yet efficient (variational) inference for probabilistic programming. We employ a smoothed interpretation of programs to obtain unbiased (reparameterisation) gradient estimators and establish technical pre-conditions by type systems. In more detail:

1. We present a simple (higher-order) programming language with conditionals. We employ *trace types* to capture precisely the samples drawn in a fully eager call-by-value evaluation strategy.
2. We endow our language with both a (measurable) denotational value semantics and a smoothed (hence approximate) value semantics. For the latter we furnish a categorical model based on Frölicher spaces.
3. We develop type systems enforcing vital technical pre-conditions: unbiasedness of the reparameterisation gradient estimator and the correctness of stochastic gradient descent, as well as the uniform convergence of the smoothing to the original problem. Thus, our smoothing approach in principle yields correct solutions up to arbitrary error tolerances.
4. We conduct an empirical evaluation demonstrating that our approach exhibits a similar convergence to an unbiased correction of the reparameterised gradient estimator by [23] – our main baseline. However our estimator is simpler and more efficient: it is faster and attains orders of magnitude reduction in work-normalised variance.

Outline. In the next section we introduce a simple higher-order probabilistic programming language, its denotational value semantics and operational semantics; Optimisation Problem 1 is then stated. Section 3 is devoted to a smoothed denotational value semantics, and we state the Smooth Optimisation Problem 2. In Sections 4 and 5 we develop annotation based type systems enforcing the correctness of SGD and the convergence of the smoothing, respectively. Related work is briefly discussed in Section 6 before we present the results of our empirical evaluation in Section 7. We conclude in Section 8 and discuss future directions.

Notation. We use the following conventions: bold font for vectors and lists, $+\!\!+$ for concatenation of lists, $\nabla_{\boldsymbol{\theta}}$ for gradients (w.r.t. $\boldsymbol{\theta}$),$[\phi]$ for the Iverson bracket of a predicate ϕ and calligraphic font for distributions, in particular $\mathcal{N}$ for normal distributions. Besides, we highlight noteworthy items using red.

2 A Simple Programming Language

In this section, we introduce our programming language, which is the simply-typed lambda calculus with reals, augmented with conditionals and sampling from continuous distributions.

2.1 Syntax

The *raw terms* of the programming language are defined by the grammar:

$$\begin{aligned} M ::= {} & x \mid \theta_i \mid \underline{r} \mid \underline{+} \mid \underline{\cdot} \mid \underline{-} \mid \underline{}^{-1} \mid \underline{\exp} \mid \underline{\log} \\ & \mid \mathbf{if}\, M < 0 \,\mathbf{then}\, M \,\mathbf{else}\, M \mid \mathbf{sample}_{\mathcal{D}} \mid \lambda x.\, M \mid M\, M \end{aligned}$$

where x and θ_i respectively range over (denumerable collections of) *variables* and *parameters*, $r \in \mathbb{R}$, and $\mathcal{D}$ is a probability distribution over $\mathbb{R}$ (potentially with a support which is a strict subset of $\mathbb{R}$). As is customary we use infix, postfix and prefix notation: $M \,\underline{+}\, N$ (addition), $M \,\underline{\cdot}\, N$ (multiplication), $M\underline{^{-1}}$ (inverse), and $\underline{-}M$ (numeric negation). We frequently omit the underline to reduce clutter.

Example 2 (Encoding the ELBO for Variational Inference). We consider the example used by [23] in their Prop. 2 to prove the biasedness of the reparameterisation gradient. (In Example 1 we discussed a simplified version thereof.) The density is

$$p(z) := \mathcal{N}(z \mid 0, 1) \cdot \begin{cases} \mathcal{N}(0 \mid -2, 1) & \text{if } z < 0 \\ \mathcal{N}(0 \mid 5, 1) & \text{otherwise} \end{cases}$$

and they use a variational family with density $q_\theta(z) := \mathcal{N}(z \mid \theta, 1)$, which is reparameterised using a standard normal noise distribution and transformation $s \mapsto s + \theta$.

First, we define an auxiliary term for the pdf of normals with mean m and standard derivation s:

$$N \equiv \lambda x, m, s.\, \left(\underline{\sqrt{2\pi}} \cdot s\right)^{-1} \cdot \underline{\exp}\left(\underline{-0.5} \cdot \left((x + (-m)) \cdot s^{-1}\right)^2\right)$$

Then, we can define

$$M \equiv \Big(\lambda z.\, \underbrace{\underline{\log}\,(N\, z\, \underline{0}\, \underline{1}) + (\mathbf{if}\ z < 0\, \mathbf{then}\, \underline{\log}\,(N\, \underline{0}\, \underline{(-2)}\, \underline{1})\, \mathbf{else}\, \underline{\log}\,(N\, \underline{0}\, \underline{5}\, \underline{1}))}_{\log p} - \underbrace{\underline{\log}\,(N\, z\, \theta\, \underline{1})}_{\log q_\theta}\Big)\, \big(\mathbf{sample}_{\mathcal{N}} + \theta\big)$$

2.2 A Basic Trace-Based Type System

Types are generated from *base types* (R and $R_{>0}$, the reals and positive reals) and *trace types* (typically Σ, which is a finite list of probability distributions) as well as by a *trace-based function space* constructor of the form $\tau \bullet \Sigma \to \tau'$. Formally types are defined by the following grammar:

trace types	$\Sigma ::= [\mathcal{D}_1, \dots, \mathcal{D}_n]$	$n \geq 0$
base types	$\iota ::= R \mid R_{>0}$	
safe types	$\sigma ::= \iota \mid \sigma \bullet [] \to \sigma$	
types	$\tau ::= \iota \mid \tau \bullet \Sigma \to \tau$	

where $\mathcal{D}_i$ are probability distributions. Intuitively a trace type is a description of the space of execution traces of a probabilistic program. Using trace types, a distinctive feature of our type system is that a program's type precisely characterises the space of its possible execution traces [24]. We use list concatenation notation $+\!\!+$ for trace types, and the shorthand $\tau_1 \to \tau_2$ for function types of the form $\tau_1 \bullet [] \to \tau_2$. Intuitively, a term has type $\tau \bullet \Sigma \to \tau'$ if, when given a value of type τ, it reduces to a value of type τ' using all the samples in Σ.

Dual context *typing judgements* of the form, $\Gamma \mid \Sigma \vdash M : \tau$, are defined in Fig. 3b, where $\Gamma = x_1 : \tau_1, \cdots, x_n : \tau_n, \theta_1 : \tau_1', \cdots, \theta_m : \tau_m'$ is a finite map describing a set of variable-type and parameter-type bindings; and the trace type Σ precisely captures the distributions from which samples are drawn in a (fully eager) call-by-value evaluation of the term M.

The subtyping of types, as defined in Fig. 3a, is essentially standard; for contexts, we define $\Gamma \sqsubseteq \Gamma'$ if for every $x : \tau$ in Γ there exists $x : \tau'$ in Γ' such that $\tau' \sqsubseteq \tau$.

Trace types are unique [18]:

Lemma 1. *If $\Gamma \mid \Sigma \vdash M : \tau$ and $\Gamma \mid \Sigma' \vdash M : \tau'$ then $\Sigma = \Sigma'$.*

A term has *safe type* σ if it does not contain $\mathbf{sample}_{\mathcal{D}}$ or σ is a base type. Thus, perhaps slightly confusingly, we have $\mid [\mathcal{D}] \vdash \mathbf{sample}_{\mathcal{D}} : R$, and R is considered a safe type. Note that we use the metavariable σ to denote safe types.

Conditionals. The branches of conditionals must have a safe type. Otherwise it would not be clear how to type terms such as

$$M \equiv \mathbf{if}\, x < 0\, \mathbf{then}\, (\lambda x.\, \mathbf{sample}_{\mathcal{N}})\, \mathbf{else}\, (\lambda x.\, \mathbf{sample}_{\mathcal{E}} + \mathbf{sample}_{\mathcal{E}})$$
$$N \equiv (\lambda f.\, f\, (f\, \mathbf{sample}_{\mathcal{N}}))\, M$$

because the branches draw a different number of samples from different distributions, and have types $R \bullet [\mathcal{N}] \to R$ and $R \bullet [\mathcal{E}, \mathcal{E}] \to R$, respectively. However, for $M' \equiv \mathbf{if}\, x < 0\, \mathbf{then}\, \mathbf{sample}_{\mathcal{N}}\, \mathbf{else}\, \mathbf{sample}_{\mathcal{E}} + \mathbf{sample}_{\mathcal{E}}$ we can (safely) type

$$\dfrac{\dfrac{x : R \mid [\mathcal{N}, \mathcal{E}, \mathcal{E}] \vdash M' : R}{\mid [] \vdash \lambda x.\, M' : R \bullet [\mathcal{N}, \mathcal{E}, \mathcal{E}] \to R}}{\mid [\mathcal{N}, \mathcal{N}, \mathcal{E}, \mathcal{E}, \mathcal{N}, \mathcal{E}, \mathcal{E}] \vdash (\lambda f.\, f\, (f\, \mathbf{sample}_{\mathcal{N}}))\, (\lambda x.\, M') : R}$$

$$\frac{}{\iota \sqsubseteq \iota} \qquad \frac{}{R_{>0} \sqsubseteq R} \qquad \frac{\tau_1' \sqsubseteq \tau_1 \quad \tau_2 \sqsubseteq \tau_2'}{(\tau_1 \bullet \Sigma \to \tau_2) \sqsubseteq (\tau_1' \bullet \Sigma \to \tau_2')}$$

(a) Subtyping

$$\frac{\Gamma \mid \Sigma \vdash M : \tau}{\Gamma' \mid \Sigma \vdash M : \tau'}\ \Gamma \sqsubseteq \Gamma', \tau \sqsubseteq \tau' \qquad \frac{}{x : \tau \mid [] \vdash x : \tau}$$

$$\frac{}{\mid [] \vdash \underline{r} : R}\ r \in \mathbb{R} \qquad \frac{}{\mid [] \vdash \underline{r} : R_{>0}}\ r \in \mathbb{R}_{>0}$$

$$\frac{}{\mid [] \vdash \underline{\circ} : R \to R \to R}\ \circ \in \{+, \cdot\} \qquad \frac{}{\mid [] \vdash \underline{\circ} : R_{>0} \to R_{>0} \to R_{>0}}\ \circ \in \{+, \cdot\}$$

$$\frac{}{\mid [] \vdash \underline{-} : R \to R} \qquad \frac{}{\mid [] \vdash \underline{\ \ }^{-1} : R_{>0} \to R_{>0}}$$

$$\frac{}{\mid [] \vdash \underline{\exp} : R \to R_{>0}} \qquad \frac{}{\mid [] \vdash \underline{\log} : R_{>0} \to R}$$

$$\frac{\Gamma \mid \Sigma \vdash L : R \quad \Gamma \mid \Sigma' \vdash M : \sigma \quad \Gamma \mid \Sigma'' \vdash N : \sigma}{\Gamma \mid \Sigma \mathbin{+\!\!+} \Sigma' \mathbin{+\!\!+} \Sigma'' \vdash \mathbf{if}\, L < 0\, \mathbf{then}\, M\, \mathbf{else}\, N : \sigma} \qquad \frac{}{\mid [\mathcal{D}] \vdash \mathbf{sample}_{\mathcal{D}} : R}$$

$$\frac{\Gamma, y : \tau_1 \mid \Sigma \vdash M : \tau_2}{\Gamma \mid [] \vdash \lambda y.\, M : \tau_1 \bullet \Sigma \to \tau_2} \qquad \frac{\Gamma \mid \Sigma_1 \vdash M : \tau_1 \bullet \Sigma_3 \to \tau_2 \quad \Gamma \mid \Sigma_2 \vdash N : \tau_1}{\Gamma \mid \Sigma_1 \mathbin{+\!\!+} \Sigma_2 \mathbin{+\!\!+} \Sigma_3 \vdash M\, N : \tau_2}$$

(b) Typing judgments

Fig. 3: A Basic Trace-based Type System

Example 3. Consider the following terms:

$$\begin{aligned} L &\equiv \lambda x.\, \mathbf{sample}_{\mathcal{N}} + \mathbf{sample}_{\mathcal{N}} \\ M &\equiv \mathbf{if}\, x < 0\, \mathbf{then}\, (\lambda y.\, y + y)\, \mathbf{sample}_{\mathcal{N}}\, \mathbf{else}\, (\mathbf{sample}_{\mathcal{N}} + \mathbf{sample}_{\mathcal{N}}) \end{aligned}$$

We can derive the following typing judgements:

$$\begin{aligned} &\mid [] \vdash L : R_{>0} \bullet [\mathcal{N}, \mathcal{N}] \to R \\ x : R_{>0} &\mid [\mathcal{N}, \mathcal{N}, \mathcal{N}] \vdash M : R \\ &\mid [] \vdash \lambda x.\, M : R_{>0} \bullet [\mathcal{N}, \mathcal{N}, \mathcal{N}] \to R \\ &\mid [\mathcal{N}, \mathcal{N}, \mathcal{N}, \mathcal{N}] \vdash (\lambda x.\, M)\, \mathbf{sample}_{\mathcal{N}} : R \\ &\mid [\mathcal{N}, \mathcal{N}] \vdash (\lambda f.\, f\, (f\, 0))\, (\lambda x.\, \mathbf{sample}_{\mathcal{N}}) : R \end{aligned}$$

Note that $\mathbf{if}\, x < 0\, \mathbf{then}\, (\lambda x.\, \mathbf{sample}_{\mathcal{N}})\, \mathbf{else}\, (\lambda x.\, x)$ is *not* typable.

2.3 Denotational Value Semantics

Next, we endow our language with a (measurable) value semantics. It is well-known that the category of measurable spaces and measurable functions is not cartesian-closed [1], which means that there is no interpretation of the lambda calculus as measurable functions. These difficulties led [14] to develop the category **QBS** of *quasi-Borel spaces*. Notably, morphisms can be combined piecewisely, which we need for conditionals.

We interpret our programming language in the category **QBS** of quasi-Borel spaces. Types are interpreted as follows:

$$[\![R]\!] \coloneqq (\mathbb{R}, M_{\mathbb{R}}) \qquad [\![R_{>0}]\!] \coloneqq (\mathbb{R}_{>0}, M_{\mathbb{R}_{>0}}) \qquad [\![[\mathcal{D}_1, \dots, \mathcal{D}_n]]\!] \coloneqq (\mathbb{R}, M_{\mathbb{R}})^n$$
$$[\![\tau_1 \bullet \Sigma \to \tau_2]\!] \coloneqq [\![\tau_1]\!] \times [\![\Sigma]\!] \Rightarrow [\![\tau_2]\!]$$

where $M_{\mathbb{R}}$ is the set of measurable functions $\mathbb{R} \to \mathbb{R}$; similarly for $M_{\mathbb{R}_{>0}}$. (As for trace types, we use list notation (and list concatenation) for traces.)

We first define a handy helper function for interpreting application. For $f : [\![\Gamma]\!] \times \mathbb{R}^{n_1} \Rightarrow [\![\tau_1 \bullet \Sigma_3 \to \tau_2]\!]$ and $g : [\![\Gamma]\!] \times \mathbb{R}^{n_2} \Rightarrow [\![\tau_1]\!]$ define

$$f \mathbin{@} g : [\![\Gamma]\!] \times \mathbb{R}^{n_1+n_2+|\Sigma_3|} \Rightarrow [\![\tau_2]\!]$$
$$(\gamma, \mathbf{s}_1 +\!\!+ \mathbf{s}_2 +\!\!+ \mathbf{s}_3) \mapsto f(\gamma, \mathbf{s}_1)(g(\gamma, \mathbf{s}_2), \mathbf{s}_3) \quad \mathbf{s}_1 \in \mathbb{R}^{n_1}, \mathbf{s}_2 \in \mathbb{R}^{n_2}, \mathbf{s}_3 \in \mathbb{R}^{|\Sigma_3|}$$

We interpret terms-in-context, $[\![\Gamma \mid \Sigma \vdash M : \tau]\!] : [\![\Gamma]\!] \times [\![\Sigma]\!] \to [\![\tau]\!]$, as follows:

$$[\![\Gamma \mid [\mathcal{D}] \vdash \mathbf{sample}_{\mathcal{D}} : R]\!](\gamma, [s]) \coloneqq s$$
$$[\![\Gamma \mid [] \vdash \lambda y.\, M : \tau_1 \bullet \Sigma \to \tau_2]\!](\gamma, []) \coloneqq$$
$$(v, \mathbf{s}) \in [\![\tau_1]\!] \times [\![\Sigma]\!] \mapsto [\![\Gamma, x : \tau_1 \mid \Sigma \vdash M : \tau_2]\!]((\gamma, v), \mathbf{s})$$
$$[\![\Gamma \mid \Sigma_1 +\!\!+ \Sigma_2 +\!\!+ \Sigma_3 \vdash M\, N : \tau]\!] \coloneqq$$
$$[\![\Gamma \mid \Sigma_1 \vdash M : \tau_1 \bullet \Sigma_3 \to \tau_2]\!] \mathbin{@} [\![\Gamma \mid \Sigma_2 \vdash N : \tau_1]\!]$$
$$[\![\Gamma \mid \Sigma_1 +\!\!+ \Sigma_2 +\!\!+ \Sigma_3 \vdash \mathbf{if}\, L < 0\, \mathbf{then}\, M\, \mathbf{else}\, N : \tau]\!](\gamma, \mathbf{s}_1 +\!\!+ \mathbf{s}_2 +\!\!+ \mathbf{s}_3)) \coloneqq$$
$$\begin{cases} [\![\Gamma \mid \Sigma_2 \vdash M : \tau]\!](\gamma, \mathbf{s}_2) & \text{if } [\![\Gamma \mid \Sigma_1 \vdash L : R]\!](\gamma, \mathbf{s}_1) < 0 \\ [\![\Gamma \mid \Sigma_3 \vdash N : \tau]\!](\gamma, \mathbf{s}_3) & \text{otherwise} \end{cases}$$

It is not difficult to see that this interpretation of terms-in-context is well-defined and total. For the conditional clause, we may assume that the trace type and the trace are presented as partitions $\Sigma_1 +\!\!+ \Sigma_2 +\!\!+ \Sigma_3$ and $\mathbf{s}_1 +\!\!+ \mathbf{s}_2 +\!\!+ \mathbf{s}_3$ respectively. This is justified because it follows from the judgement $\Gamma \mid \Sigma_1 +\!\!+ \Sigma_2 +\!\!+ \Sigma_3 \vdash \mathbf{if}\, L < 0\, \mathbf{then}\, M\, \mathbf{else}\, N : \tau$ that $\Gamma \mid \Sigma_1 \vdash L : R$, $\Gamma \mid \Sigma_2 \vdash M : \sigma$ and $\Gamma \mid \Sigma_3 \vdash N : \sigma$ are provable; and we know that each of Σ_1, Σ_2 and Σ_3 is unique, thanks to Lemma 1; their respective lengths then determine the partition $\mathbf{s}_1 +\!\!+ \mathbf{s}_2 +\!\!+ \mathbf{s}_3$. Similarly for the application clause, the components Σ_1 and Σ_2 are determined by Lemma 1, and Σ_3 by the type of M.

2.4 Relation to Operational Semantics

We can also endow our language with a big-step CBV sampling-based semantics similar to [7,26], as defined in [18, Fig. 6]. We write $M \Downarrow_w^{\mathbf{s}} V$ to mean that M reduces to value V, which is a real constant or an abstraction, using the execution trace $\mathbf{s}$ and accumulating weight w.

Based on this, we can define the *value*- and *weight*-functions:

$$\mathrm{value}_M(\mathbf{s}) := \begin{cases} V & \text{if } M \Downarrow_w^{\mathbf{s}} V \\ \mathrm{undef} & \text{otherwise} \end{cases} \qquad \mathrm{weight}_M(\mathbf{s}) := \begin{cases} w & \text{if } M \Downarrow_w^{\mathbf{s}} V \\ 0 & \text{otherwise} \end{cases}$$

Our semantics is a bit non-standard in that for conditionals we evaluate both branches eagerly. The technical advantage is that for every (closed) term-in-context, $\mid [\mathcal{D}_1, \cdots, \mathcal{D}_n] \vdash M : \iota$, M reduces to a (unique) value using exactly the traces of the length encoded in the typing, i.e., n.

So in this sense, the operational semantics is "total": there is no divergence. Notice that there is no partiality caused by partial primitives such as $1/x$, thanks to the typing.

Moreover there is a simple connection to our denotational value semantics:

Proposition 1. *Let* $\mid [\mathcal{D}_1, \ldots, \mathcal{D}_n] \vdash M : \iota$. *Then*

1. $\mathrm{dom}(\mathrm{value}_M) = \mathbb{R}^n$
2. $\underline{[\![M]\!]} = \mathrm{value}_M$
3. $\mathrm{weight}_M(\mathbf{s}) = \prod_{j=1}^n \mathrm{pdf}_{\mathcal{D}_j}(s_j)$

2.5 Problem Statement

We are finally ready to formally state our optimisation problem:

Problem 1. Optimisation

Given: term-in-context, $\theta_1 : \iota_1, \cdots, \theta_m : \iota_m \mid [\mathcal{D}_1, \ldots, \mathcal{D}_n] \vdash M : R$

Find: $\mathrm{argmin}_{\boldsymbol{\theta}}\ \mathbb{E}_{s_1 \sim \mathcal{D}_1, \ldots, s_n \sim \mathcal{D}_n} [[\![M]\!](\boldsymbol{\theta}, \mathbf{s})]$

3 Smoothed Denotational Value Semantics

Now we turn to our smoothed denotational value semantics, which we use to avoid the bias in the reparameterisation gradient estimator. It is parameterised by a family of smooth functions $\sigma_\eta : \mathbb{R} \to [0, 1]$. Intuitively, we replace the Heaviside step-function arising in the interpretation of conditionals by smooth approximations (cf. Fig. 2). In particular, conditionals $\mathbf{if}\, z < 0\, \mathbf{then}\, \underline{0}\, \mathbf{else}\, \underline{1}$ are interpreted as $z \mapsto \sigma_\eta(z)$ rather than $[z \geq 0]$ (using Iverson brackets).

Our primary example is $\sigma_\eta(x) := \sigma(\frac{x}{\eta})$, where σ is the (logistic) sigmoid $\sigma(x) := \frac{1}{1+\exp(-x)}$, see Fig. 2. Whilst at this stage no further properties other than smoothness are required, we will later need to restrict σ_η to have good properties, in particular to convergence to the Heaviside step function.

As a categorical model we propose *vector Frölicher spaces* **VectFr**, which (to our knowledge) is a new construction, affording a simple and direct interpretation of the smoothed conditionals.

3.1 Frölicher Spaces

We recall the definition of Frölicher spaces, which generalise smooth spaces[4]: A *Frölicher space* is a triple $(X, \mathcal{C}_X, \mathcal{F}_X)$ where X is a set, $\mathcal{C}_X \subseteq \mathbf{Set}(\mathbb{R}, X)$ is a set of *curves* and $\mathcal{F}_X \subseteq \mathbf{Set}(X, \mathbb{R})$ is a set of *functionals*. satisfying

1. if $c \in \mathcal{C}_X$ and $f \in \mathcal{F}_X$ then $f \circ c \in C^\infty(\mathbb{R}, \mathbb{R})$
2. if $c : \mathbb{R} \to X$ such that for all $f \in \mathcal{F}_X$, $f \circ c \in C^\infty(\mathbb{R}, \mathbb{R})$ then $c \in \mathcal{C}_X$
3. if $f : X \to \mathbb{R}$ such that for all $c \in \mathcal{C}_X$, $f \circ c \in C^\infty(\mathbb{R}, \mathbb{R})$ then $f \in \mathcal{F}_X$.

A *morphism* between Frölicher spaces $(X, \mathcal{C}_X, \mathcal{F}_X)$ and $(Y, \mathcal{C}_Y, \mathcal{F}_Y)$ is a map $\phi : X \to Y$ satisfying $f \circ \phi \circ c \in C^\infty(\mathbb{R}, \mathbb{R})$ for all $f \in \mathcal{F}_Y$ and $c \in \mathcal{C}_X$.

Frölicher spaces and their morphisms constitute a category **Fr**, which is well-known to be cartesian closed [13,33].

3.2 Vector Frölicher Spaces

To interpret our programming language smoothly we would like to interpret conditionals as σ_η-weighted convex combinations of its branches:

$$\begin{aligned}&[\![\mathbf{if}\, L < 0\, \mathbf{then}\, M\, \mathbf{else}\, N]\!]_\eta(\gamma, \mathbf{s}_1 +\!\!+ \mathbf{s}_2 +\!\!+ \mathbf{s}_3) :=\\ &\qquad \sigma_\eta(-[\![L]\!]_\eta(\gamma, \mathbf{s}_1)) \cdot [\![M]\!]_\eta(\gamma, \mathbf{s}_2) + \sigma_\eta([\![L]\!]_\eta(\gamma, \mathbf{s}_1)) \cdot [\![N]\!]_\eta(\gamma, \mathbf{s}_3) \qquad (4)\end{aligned}$$

By what we have discussed so far, this only makes sense if the branches have ground type because Frölicher spaces are not equipped with a vector space structure but we take weighted combinations of morphisms. In particular if $\phi_1, \phi_2 : X \to Y$ and $\alpha : X \to \mathbb{R}$ are morphisms then $\alpha\, \phi_1 + \phi_2$ ought to be a morphism too. Therefore, we enrich Frölicher spaces with an additional vector space structure:

Definition 1. *An* $\mathbb{R}$-vector Frölicher space *is a Frölicher space* $(X, \mathcal{C}_X, \mathcal{F}_X)$ *such that* X *is an* $\mathbb{R}$*-vector space and whenever* $c, c' \in \mathcal{C}_X$ *and* $\alpha \in C^\infty(\mathbb{R}, \mathbb{R})$ *then* $\alpha\, c + c' \in \mathcal{C}_X$ *(defined pointwise).*

A morphism *between* $\mathbb{R}$-vector Frölicher spaces *is a morphism between Frölicher spaces, i.e.* $\phi : (X, \mathcal{C}_X, \mathcal{F}_X) \to (Y, \mathcal{C}_Y, \mathcal{F}_Y)$ *is a morphism if for all* $c \in \mathcal{C}_X$ *and* $f \in \mathcal{F}_Y$*,* $f \circ \phi \circ c \in C^\infty(\mathbb{R}, \mathbb{R})$*.*

$\mathbb{R}$-*vector Frölicher space* and their morphisms constitute a category **VectFr**. There is an evident forgetful functor fully faithfully embedding **VectFr** in **Fr**. Note that the above restriction is a bit stronger than requiring that $\mathcal{C}_X$ is also a vector space. (α is not necessarily a constant.) The main benefit is the following, which is crucial for the interpretation of conditionals as in Eq. (4):

Lemma 2. *If* $\phi_1, \phi_2 \in \mathbf{VectFr}(X, Y)$ *and* $\alpha \in \mathbf{VectFr}(X, \mathbb{R})$ *then* $\alpha\, \phi_1 + \phi_2 \in \mathbf{VectFr}(X, Y)$ *(defined pointwisely).*

Proof. Suppose $c \in \mathcal{C}_X$ and $f \in \mathcal{F}_Y$. Then $(\alpha_1\, \phi_1 + \phi_2) \circ c = (\alpha \circ c) \cdot (\phi_1 \circ c) + (\phi_2 \circ c) \in \mathcal{C}_Y$ (defined pointwisely) and the claim follows.

[4] $C^\infty(\mathbb{R}, \mathbb{R})$ is the set of smooth functions $\mathbb{R} \to \mathbb{R}$

Similarly as for Frölicher spaces, if X is an $\mathbb{R}$-vector space then any $\mathcal{C} \subseteq$ **Set**$(X, \mathbb{R})$ *generates* a $\mathbb{R}$-vector Frölicher space $(X, \mathcal{C}_X, \mathcal{F}_X)$, where

$$\begin{aligned}
\mathcal{F}_X &\coloneqq \{f : X \to \mathbb{R} \mid \forall c \in \mathcal{C}.\, f \circ c \in C^\infty(\mathbb{R}, \mathbb{R})\}\\
\widetilde{\mathcal{C}}_X &\coloneqq \{c : \mathbb{R} \to X \mid \forall f \in \mathcal{F}_X.\, f \circ c \in C^\infty(\mathbb{R}, \mathbb{R})\}\\
\mathcal{C}_X &\coloneqq \left\{\sum_{i=1}^{n} \alpha_i\, c_i \mid n \in \mathbb{N}, \forall i \leq n.\, \alpha_i \in C^\infty(\mathbb{R}, \mathbb{R}), c_i \in \widetilde{\mathcal{C}}_X\right\}
\end{aligned}$$

Having modified the notion of Frölicher spaces generated by a set of curves, the proof for cartesian closure carries over [18] and we conclude:

Proposition 2. **VectFr** *is cartesian closed.*

3.3 Smoothed Interpretation

We have now discussed all ingredients to interpret our language (smoothly) in the cartesian closed category **VectFr**. We call $[\![M]\!]_\eta$ the η-*smoothing* of $[\![M]\!]$ (or of M, by abuse of language). The interpretation is mostly standard and follows Section 2.3, except for the case for conditionals. The latter is given by Eq. (4), for which the additional vector space structure is required.

Finally, we can phrase a smoothed version of our Optimisation Problem 1:

Problem 2. η-Smoothed Optimisation

Given: term-in-context, $\theta_1 : \iota_1, \cdots, \theta_m : \iota_m \mid [\mathcal{D}_1, \ldots, \mathcal{D}_n] \vdash M : R$, and *accuracy* coefficient $\eta > 0$

Find: $\operatorname{argmin}_{\boldsymbol{\theta}}\ \mathbb{E}_{s_1 \sim \mathcal{D}_1, \ldots, s_n \sim \mathcal{D}_n}\left[[\![M]\!]_\eta(\boldsymbol{\theta}, \mathbf{s})\right]$

4 Correctness of SGD for Smoothed Problem and Unbiasedness of the Reparameterisation Gradient

Next, we apply stochastic gradient descent (SGD) with the reparameterisation gradient estimator to the smoothed problem (for the batch size $N = 1$):

$$\boldsymbol{\theta}_{k+1} \coloneqq \boldsymbol{\theta}_k - \gamma_k \cdot \nabla_\theta [\![M]\!]_\eta\,(\boldsymbol{\theta}_k, \mathbf{s}_k) \qquad\qquad \mathbf{s}_k \sim \mathcal{D} \tag{5}$$

where $\boldsymbol{\theta} \mid [\mathbf{s} \sim \mathcal{D}] \vdash M : R$ (slightly abusing notation in the trace type).

A classical choice for the step-size sequence is $\gamma_k \in \Theta(1/k)$, which satisfies the so-called *Robbins-Monro* criterion:

$$\sum_{k \in \mathbb{N}} \gamma_k = \infty \qquad\qquad \sum_{k \in \mathbb{N}} \gamma_k^2 < \infty \tag{6}$$

In this section we wish to establish the correctness of the SGD procedure applied to the smoothing Eq. (5).

4.1 Desiderata

First, we ought to take a step back and observe that the optimisation problems we are trying to solve can be ill-defined due to a failure of integrability: take $M \equiv (\lambda x.\, \underline{\exp}\,(x \,\underline{\cdot}\, x))\, \mathbf{sample}_{\mathcal{N}}$: we have $\mathbb{E}_{z\sim\mathcal{N}}[[\![M]\!](z)] = \infty$, independently of parameters. Therefore, we aim to guarantee:

(SGD0) The optimisation problems (both smoothed and unsmoothed) are well-defined.

Since $\mathbb{E}[[\![M]\!]_\eta(\boldsymbol{\theta}, \mathbf{s})]$ (and $\mathbb{E}[[\![M]\!](\boldsymbol{\theta}, \mathbf{s})]$) may not be a convex function in the parameters $\boldsymbol{\theta}$, we cannot hope to always find global optima. We seek instead *stationary points*, where the gradient w.r.t. the parameters $\boldsymbol{\theta}$ vanishes. The following results (whose proof is standard) provide sufficient conditions for the convergence of SGD to stationary points (see e.g. [3] or [2, Chapter 2]):

Proposition 3 (Convergence). *Suppose* $(\gamma_k)_{k\in\mathbb{N}}$ *satisfies the Robbins-Monro criterion Eq.* (6) *and* $g(\boldsymbol{\theta}) := \mathbb{E}_{\mathbf{s}}[f(\boldsymbol{\theta}, \mathbf{s})]$ *is well-defined. If* $\boldsymbol{\Theta} \subseteq \mathbb{R}^m$ *satisfies*

(SGD1) Unbiasedness: $\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{s}}[\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}, \mathbf{s})]$ *for all* $\boldsymbol{\theta} \in \boldsymbol{\Theta}$
(SGD2) g *is L-Lipschitz smooth on* $\boldsymbol{\Theta}$ *for some* $L > 0$*:*

$$\|\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}) - \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}')\| \leq L \cdot \|\boldsymbol{\theta} - \boldsymbol{\theta}'\| \qquad \text{for all } \boldsymbol{\theta}, \boldsymbol{\theta}' \in \boldsymbol{\Theta}$$

(SGD3) Bounded Variance: $\sup_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \mathbb{E}_{\mathbf{s}}[\|\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}, \mathbf{s})\|^2] < \infty$

then $\inf_{i\in\mathbb{N}} \mathbb{E}[\|\nabla g(\boldsymbol{\theta}_i)\|^2] = 0$ *or* $\boldsymbol{\theta}_i \notin \boldsymbol{\Theta}$ *for some* $i \in \mathbb{N}$.

Unbiasedness (SGD1) requires commuting differentiation and integration. The validity of this operation can be established by the dominated convergence theorem [21, Theorem 6.28], see [18]. To be applicable the partial derivatives of f w.r.t. the parameters need to be dominated uniformly by an integrable function. Formally:

Definition 2. *Let* $f : \boldsymbol{\Theta} \times \mathbb{R}^n \to \mathbb{R}$ *and* $g : \mathbb{R}^n \to \mathbb{R}$. *We say that* g uniformly dominates f *if for all* $(\boldsymbol{\theta}, \mathbf{s}) \in \boldsymbol{\Theta} \times \mathbb{R}^n$, $|f(\boldsymbol{\theta}, \mathbf{s})| \leq g(\mathbf{s})$.

Also note that for Lipschitz smoothness (SGD2) it suffices to uniformly bound the second-order partial derivatives.

In the remainder of this section we present two type systems which restrict the language to guarantee properties (SGD0) to (SGD3).

4.2 Piecewise Polynomials and Distributions with Finite Moments

As a first illustrative step we consider a type system $\vdash_{\text{poly}}$, which restricts terms to (piecewise) polynomials, and distributions with finite moments. Recall that a distribution $\mathcal{D}$ has (all) *finite moments* if for all $p \in \mathbb{N}$, $\mathbb{E}_{s\sim\mathcal{D}}[|s|^p] < \infty$. Distributions with finite moments include the following commonly used distributions: normal, exponential, logistic and gamma distributions. A non-example is the Cauchy distribution, which famously does not even have an expectation.

Definition 3. *For a distribution $\mathcal{D}$ with finite moments, $f : \mathbb{R}^n \to \mathbb{R}$ has (all)* finite moments *if for all $p \in \mathbb{N}$, $\mathbb{E}_{\mathbf{s}\sim\mathcal{D}}[|f(\mathbf{s})|^p] < \infty$.*

Functions with finite moments have good closure properties:

Lemma 3. *If $f, g : \mathbb{R}^n \to \mathbb{R}$ have (all) finite moments so do $-f, f + g, f \cdot g$.*

In particular, if a distribution has finite moments then polynomials do, too. Consequently, intuitively, it is sufficient to simply (the details are explicitly spelled out in [18]):

1. require that the distributions $\mathcal{D}$ in the sample rule have finite moments:

$$\frac{}{\mid [\mathcal{D}] \vdash_{\text{poly}} \mathbf{sample}_{\mathcal{D}} : R} \ \mathcal{D} \text{ has finite moments}$$

2. remove the rules for $\underline{}^{-1}$, $\underline{\exp}$ and $\underline{\log}$ from the type system $\vdash_{\text{poly}}$.

Type Soundness I: Well-Definedness. Henceforth, we fix parameters $\theta_1 : \iota_1, \ldots, \theta_m : \iota_m$. Intuitively, it is pretty obvious that $[\![M]\!]$ is a piecewise polynomial whenever $\boldsymbol{\theta} \mid \Sigma \vdash_{\text{poly}} M : \iota$. Nonetheless, we prove the property formally to illustrate our proof technique, a variant of logical relations, employed throughout the rest of the paper.

We define a slightly stronger logical predicate $\mathcal{P}_\tau^{(n)}$ on $\boldsymbol{\Theta} \times \mathbb{R}^n \to [\![\tau]\!]$, which allows us to obtain a uniform upper bound:

1. $f \in \mathcal{P}_\iota^{(n)}$ if f is uniformly dominated by a function with finite moments
2. $f \in \mathcal{P}_{\tau_1 \bullet \Sigma_3 \to \tau_2}^{(n)}$ if for all $n_2 \in \mathbb{N}$ and $g \in \mathcal{P}_{\tau_1}^{(n+n_2)}$, $f \odot g \in \mathcal{P}_{\tau_2}^{(n+n_2+|\Sigma_3|)}$

where for $f : \boldsymbol{\Theta} \times \mathbb{R}^{n_1} \to [\![\tau_1 \bullet \Sigma_3 \to \tau_2]\!]$ and $g : \boldsymbol{\Theta} \times \mathbb{R}^{n_1+n_2} \to [\![\tau_1]\!]$ we define

$$\begin{aligned} f \odot g : \boldsymbol{\Theta} \times \mathbb{R}^{n_1+n_2+|\Sigma_3|} &\to \tau_2 \\ (\boldsymbol{\theta}, \mathbf{s}_1 \mathbin{+\!\!+} \mathbf{s}_2 \mathbin{+\!\!+} \mathbf{s}_3) &\mapsto f(\boldsymbol{\theta}, \mathbf{s}_1)(g(\boldsymbol{\theta}, \mathbf{s}_1 \mathbin{+\!\!+} \mathbf{s}_2), \mathbf{s}_3) \end{aligned}$$

Intuitively, g may depend on the samples in $\mathbf{s}_2$ (in addition to $\mathbf{s}_1$) and the function application may consume further samples $\mathbf{s}_3$ (as determined by the trace type Σ_3). By induction on safe types we prove the following result, which is important for conditionals:

Lemma 4. *If $f \in \mathcal{P}_\iota^{(n)}$ and $g, h \in \mathcal{P}_\sigma^{(n)}$ then $[f(-) < 0] \cdot g + [f(-) \geq 0] \cdot h \in \mathcal{P}_\sigma^{(n)}$.*

Proof. For base types it follows from Lemma 3. Hence, suppose σ has the form $\sigma_1 \bullet [] \to \sigma_2$. Let $n_2 \in \mathbb{N}$ and $x \in \mathcal{P}_{\sigma_1}^{n+n_2}$. By definition, $(g \odot x), (h \odot x) \in \mathcal{P}_{\sigma_2}^{(n+n_2)}$. Let $\widehat{f}$ be the extension (ignoring the additional samples) of f to $\boldsymbol{\Theta} \times \mathbb{R}^{n+n_2} \to \mathbb{R}$. It is easy to see that also $\widehat{f} \in \mathcal{P}_\iota^{(n+n_2)}$ By the inductive hypothesis,

$$[\widehat{f}(-) < 0] \cdot (g \odot x) + [\widehat{f}(-) \geq 0] \cdot (h \odot x) \in \mathcal{P}_{\sigma_2}^{(n+n_2)}$$

Finally, by definition,

$$([f(-) < 0] \cdot g + [f(-) \geq 0] \cdot h) \odot x = [\widehat{f}(-) < 0] \cdot (g \odot x) + [\widehat{f}(-) \geq 0] \cdot (h \odot x)$$

Assumption 1 *We assume that $\boldsymbol{\Theta} \subseteq [\![\iota_1]\!] \times \cdots \times [\![\iota_m]\!]$ is compact.*

Lemma 5 (Fundamental). *If $\boldsymbol{\theta}, x_1 : \tau_1, \ldots, x_\ell : \tau_\ell \mid \Sigma \vdash_{\text{poly}} M : \tau$, $n \in \mathbb{N}$, $\xi_1 \in \mathcal{P}_{\tau_1}^{(n)}, \ldots, \xi_\ell \in \mathcal{P}_{\tau_\ell}^{(n)}$ then $[\![M]\!] * \langle \xi_1, \ldots, \xi_\ell \rangle \in \mathcal{P}_\tau^{(n+|\Sigma|)}$, where*

$$\begin{aligned}[\![M]\!] * \langle \xi_1, \ldots, \xi_\ell \rangle : \boldsymbol{\Theta} \times \mathbb{R}^{n+|\Sigma|} &\to [\![\tau]\!] \\ (\boldsymbol{\theta}, \mathbf{s} +\!\!+ \mathbf{s}') &\mapsto [\![M]\!]((\boldsymbol{\theta}, \xi_1(\boldsymbol{\theta}, \mathbf{s}), \ldots, \xi_\ell(\boldsymbol{\theta}, \mathbf{s})), \mathbf{s}')\end{aligned}$$

It is worth noting that, in contrast to more standard fundamental lemmas, here we need to capture the dependency of the free variables on some number n of further samples. E.g. in the context of $(\lambda x.\, x)\, \mathbf{sample}_{\mathcal{N}}$ the subterm x depends on a sample although this is not apparent if we consider x in isolation.

Lemma 5 is proven by structural induction [18]. The most interesting cases include: parameters, primitive operations and conditionals. In the case for parameters we exploit the compactness of $\boldsymbol{\Theta}$ (Assumption 1). For primitive operations we note that as a consequence of Lemma 3 each $\mathcal{P}_\iota^{(n)}$ is closed under negation[5], addition and multiplication. Finally, for conditionals we exploit Lemma 3.

Type Soundness II: Correctness of SGD. Next, we address the integrability for the *smoothed* problem as well as (SGD1) to (SGD3). We establish that not only $[\![M]\!]_\eta$ but also its partial derivatives up to order 2 are uniformly dominated by functions with finite moments. For this to possibly hold we require:

Assumption 2 *For every $\eta > 0$,*

$$\sup_{x \in \mathbb{R}} |\sigma_\eta(x)| < \infty \qquad \sup_{x \in \mathbb{R}} |\sigma_\eta'(x)| < \infty \qquad \sup_{x \in \mathbb{R}} |\sigma_\eta''(x)| < \infty$$

Note that, for example, the logistic sigmoid satisfies Assumption 2.

We can then prove a fundamental lemma similar to Lemma 5, *mutatis mutandis*, using a logical predicate in **VectFr**. We stipulate $f \in \mathcal{Q}_\iota^{(n)}$ if its partial derivatives up to order 2 are uniformly dominated by a function with finite moments. In addition to Lemma 3 we exploit standard rules for differentiation (such as the sum, product and chain rule) as well as Assumption 2. We conclude:

Proposition 4. *If $\boldsymbol{\theta} \mid \Sigma \vdash_{\text{poly}} M : R$ then the partial derivatives up to order 2 of $[\![M]\!]_\eta$ are uniformly dominated by a function with all finite moments.*

Consequently, the Smoothed Optimisation Problem 2 is not only well-defined but, by the dominated convergence theorem [21, Theorem 6.28], the reparameterisation gradient estimator is unbiased. Furthermore, (SGD1) to (SGD3) are satisfied and SGD is correct.

Discussion. The type system $\vdash_{\text{poly}}$ is simple yet guarantees correctness of SGD. However, it is somewhat restrictive; in particular, it does not allow the expression of many ELBOs arising in variational inference directly as they often have the form of logarithms of exponential terms (cf. Example 2).

[5] for $\iota = R$

4.3 A Generic Type System with Annotations

Next, we present a generic type system with annotations. In Section 4.4 we give an instantiation to make $\vdash_{\text{poly}}$ more permissible and in Section 5 we turn towards a different property: the uniform convergence of the smoothings.

Typing judgements have the form $\Gamma \mid \Sigma \vdash_? M : \tau$, where "?" indicates the property we aim to establish, and we annotate base types. Thus, types are generated from

$$\begin{array}{ll} \textbf{trace types} & \Sigma ::= [s_1 \sim \mathcal{D}_1, \dots, s_n \sim \mathcal{D}_n] \\ \textbf{base types} & \iota ::= R \mid R_{>0} \\ \textbf{safe types} & \sigma ::= \iota^{\beta} \mid \sigma \bullet [] \to \sigma \\ \textbf{types} & \tau ::= \iota^{\alpha} \mid \tau \bullet \Sigma \to \tau \end{array}$$

Annotations are drawn from a set and may possibly restricted for safe types. Secondly, the trace types are now annotated with variables, typically $\Sigma = [s_1 \sim \mathcal{D}_1, \dots, s_n \sim \mathcal{D}_n]$ where the variables s_j are pairwise distinct.

For the subtyping relation we can constrain the annotations at the base type level [18]; the extension to higher types is accomplished as before.

The typing rules have the same form but they are extended with the annotations on base types and side conditions possibly constraining them. For example, the rules for addition, exponentiation and sampling are modified as follows:

$$\frac{}{\mid [] \vdash_? \underline{+} : \iota^{\alpha_1} \to \iota^{\alpha_2} \to \iota^{\alpha}}\ \text{(cond. Add)} \qquad \frac{}{\mid [] \vdash_? \underline{\exp} : R^{\alpha} \to R^{\alpha'}_{>0}}\ \text{(cond. Exp)}$$

$$\frac{}{\mid [s_j \sim \mathcal{D}] \vdash_? \mathbf{sample}_{\mathcal{D}} : R^{\alpha}}\ \text{(cond. Sample)}$$

The rules for subtyping, variables, abstractions and applications do not need to be changed at all but they use annotated types instead of the types of Section 2.2.

$$\frac{\Gamma \mid \Sigma \vdash_? M : \tau}{\Gamma' \mid \Sigma \vdash_? M : \tau'}\ \Gamma \sqsubseteq_? \Gamma', \tau \sqsubseteq_? \tau' \qquad \frac{}{x : \tau \mid [] \vdash_? x : \tau}$$

$$\frac{\Gamma, y : \tau_1 \mid \Sigma \vdash_? M : \tau_2}{\Gamma \mid [] \vdash_? \lambda y.\, M : \tau_1 \bullet \Sigma \to \tau_2} \qquad \frac{\Gamma \mid \Sigma_2 \vdash_? M : \tau_1 \bullet \Sigma_3 \to \tau_2 \quad \Gamma \mid \Sigma_1 \vdash_? N : \tau_1}{\Gamma \mid \Sigma_1 +\!\!+ \Sigma_2 +\!\!+ \Sigma_3 \vdash_? M\, N : \tau_2}$$

The full type system is presented in [18].

$\vdash_{\text{poly}}$ can be considered a special case of $\vdash_?$ whereby we use the singleton $*$ as annotations, a contradictory side condition (such as false) for the undesired primitives $\underline{^{-1}}$, $\underline{\exp}$ and $\underline{\log}$, and use the side condition "$\mathcal{D}$ has finite moments" for sample as above.

Table 1 provides an overview of the type systems of this paper and their purpose. $\vdash_?$ and its instantiations refine the basic type system of Section 2.2 in the sense that if a term-in-context is provable in the annotated type system, then its erasure (i.e. erasure of the annotations of base types and distributions) is provable in the basic type system. This is straightforward to check.

Table 1: Overview of type systems in this paper.

property	Section	judgement	annotation
totality	Section 2.2	$\vdash$	–
correctness SGD	Section 4.2	$\vdash_{\text{poly}}$	none/$*$
	Section 4.4	$\vdash_{\text{SGD}}$	0/1
uniform convergence	Section 5.1	$\vdash_{\text{unif}}$	$(\mathbf{f}, \Delta)/(\mathbf{t}, \Delta)$

$$\frac{}{\mid [] \vdash_{\text{SGD}} \underline{\exp} : R^{(0)} \to R^{(1)}_{>0}} \qquad \frac{}{\mid [] \vdash_{\text{SGD}} \underline{\log} : R^{(e)}_{>0} \to R^{(0)}}$$

$$\frac{}{\mid [] \vdash_{\text{SGD}} \underline{+} : \iota^{(0)} \to \iota^{(0)} \to \iota^{(0)}} \qquad \frac{}{\mid [] \vdash_{\text{SGD}} \underline{\cdot} : \iota^{(e)} \to \iota^{(e)} \to \iota^{(e)}}$$

$$\frac{}{\mid [] \vdash_{\text{SGD}} \underline{-} : R^{(0)} \to R^{(0)}} \qquad \frac{}{\mid [] \vdash_{\text{SGD}} \underline{^{-1}} : R^{(e)}_{>0} \to R^{(e)}_{>0}}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\text{SGD}} L : \iota^{(0)} \quad \Gamma \mid \Sigma' \vdash_{\text{SGD}} M : \sigma \quad \Gamma \mid \Sigma'' \vdash_{\text{SGD}} N : \sigma}{\Gamma \mid \Sigma +\!\!+ \Sigma' +\!\!+ \Sigma'' \vdash_{\text{SGD}} \mathbf{if}\, L < 0 \,\mathbf{then}\, M \,\mathbf{else}\, N : \sigma}$$

$$\frac{}{\mid [s_j \sim \mathcal{D}] \vdash_{\text{SGD}} \mathbf{sample}_{\mathcal{D}} : R^{(0)}}\ \mathcal{D} \text{ has finite moments}$$

Fig. 4: Excerpt of the typing rules (cf. [18]) for the correctness of SGD.

4.4 A More Permissible Type System

In this section we discuss another instantiation, $\vdash_{\text{SGD}}$, of the generic type system system to guarantee (SGD0) to (SGD3), which is more permissible than $\vdash_{\text{poly}}$. In particular, we would like to support Example 2, which uses logarithms and densities involving exponentials. Intuitively, we need to ensure that subterms involving $\underline{\exp}$ are "neutralised" by a corresponding $\underline{\log}$. To achieve this we annotate base types with 0 or 1, ordered discretely. 0 is the only annotation for safe base types and can be thought of as "integrable"; 1 denotes "needs to be passed through log". More precisely, we constrain the typing rules such that if $\boldsymbol{\theta} \mid \Sigma \vdash_{\text{SGD}} M : \iota^{(e)}$ then[6] $\log^e \circ [\![M]\!]$ and the partial derivatives of $\log^e \circ [\![M]\!]_\eta$ up to order 2 are uniformly dominated by a function with finite moments.

We subtype base types as follows: $\iota_1^{(e_1)} \sqsubseteq_{\text{SGD}} \iota_2^{(e_2)}$ if $\iota_1 \sqsubseteq \iota_2$ (as defined in Fig. 3a) and $e_1 = e_2$, or $\iota_1 = R_{>0} = \iota_2$ and $e_1 \leq e_2$. The second disjunct may come as a surprise but we ensure that terms of type $R^{(0)}_{>0}$ cannot depend on samples at all.

In Fig. 4 we list the most important rules; we relegate the full type system to [18]. $\underline{\exp}$ and $\underline{\log}$ increase and decrease the annotation respectively. The rules for the primitive operations and conditionals are motivated by the closure properties

[6] using the convention $\log^0$ is the identity

of Lemma 3 and the elementary fact that $\log\circ(f\cdot g) = (\log\circ f) + (\log\circ g)$ and $\log\circ(f^{-1}) = -\log\circ f$ for $f, g : \boldsymbol{\Theta}\times\mathbb{R}^n \to \mathbb{R}$.

Example 4. $\theta : R^{(0)}_{>0} \mid [\mathcal{N}, \mathcal{N}] \vdash_{\mathrm{SGD}} \underline{\log}\,(\theta^{-1}\cdot\underline{\exp}\,(\mathbf{sample}_{\mathcal{N}})) + \mathbf{sample}_{\mathcal{N}} : R^{(0)}$

Note that the branches of conditionals need to have safe type, which rules out branches with type $R^{(1)}$. This is because logarithms do not behave nicely when composed with addition as used in the smoothed interpretation of conditionals.

Besides, observe that in the rules for logarithm and inverses $e = 0$ is allowed, which may come as a surprise[7]. This is e.g. necessary for the typability of the variational inference Example 2:

Example 5 (Typing for Variational Inference). It holds $\mid [] \vdash N : R^{(0)} \to R^{(0)} \to R^{(0)}_{>0} \to R^{(1)}_{>0}$ and $\theta : R^{(0)} \mid [s_1 \sim \mathcal{N}] \vdash M : R^{(0)}$.

Type Soundness. To formally establish type soundness, we can use a logical predicate, which is very similar to the one in Section 4.2 (N.B. the additional Item 2): in particular $f \in \mathcal{Q}^{(n)}_{\iota^{(e)}}$ if

1. partial derivatives of $\log^e\circ f$ up to order 2 are uniformly dominated by a function with finite moments
2. if $\iota^{(e)}$ is $R^{(0)}_{>0}$ then f is dominated by a positive constant function

Using this and a similar logical predicate for $[\![(-)]\!]$ we can show:

Proposition 5. *If* $\theta_1 : \iota^{(0)}, \ldots, \theta_m : \iota^{(0)}_m \mid \Sigma \vdash_{\mathrm{SGD}} M : \iota^{(0)}$ *then*

1. *all distributions in* Σ *have finite moments*
2. $[\![M]\!]$ *and for each* $\eta > 0$ *the partial derivatives up to order 2 of* $[\![M]\!]_\eta$ *are uniformly dominated by a function with finite moments.*

Consequently, again the Smoothed Optimisation Problem 2 is not only well-defined but by the dominated convergence theorem, the reparameterisation gradient estimator is unbiased. Furthermore, (SGD1) to (SGD3) are satisfied and SGD is correct.

5 Uniform Convergence

In the preceding section we have shown that SGD with the reparameterisation gradient can be employed to correctly (in the sense of Proposition 3) solve the Smoothed Optimisation Problem 2 for any fixed accuracy coefficient. However, *a priori*, it is not clear how a solution of the Smoothed Problem 2 can help to solve the original Problem 1.

The following illustrates the potential for significant discrepancies:

[7] Recall that terms of type $R^{(0)}_{>0}$ cannot depend on samples.

Example 6. Consider $M \equiv \mathbf{if}\, 0 < 0\, \mathbf{then}\, \theta \cdot \theta + \underline{1}\, \mathbf{else}\, (\theta - \underline{1}) \cdot (\theta - \underline{1})$. Notice that the global minimum and the only stationary point of $[\![M]\!]_\eta$ is at $\theta = \frac{1}{2}$ regardless of $\eta > 0$, where $[\![M]\!]_\eta(\frac{1}{2}) = \frac{3}{4}$. On the other hand $[\![M]\!](\frac{1}{2}) = \frac{1}{4}$ and the global minimum of $[\![M]\!]$ is at $\theta = 1$.

In this section we investigate under which conditions the smoothed objective function converges to the original objective function *uniformly* in $\boldsymbol{\theta} \in \boldsymbol{\Theta}$:

$$\text{(Unif)}\quad \mathbb{E}_{\mathbf{s}\sim\mathcal{D}}\left[[\![M]\!]_\eta(\boldsymbol{\theta}, \mathbf{s})\right] \xrightarrow{\text{unif.}} \mathbb{E}_{\mathbf{s}\sim\mathcal{D}}\left[[\![M]\!](\boldsymbol{\theta}, \mathbf{s})\right] \text{ as } \eta \searrow 0 \text{ for } \boldsymbol{\theta} \in \boldsymbol{\Theta}$$

We design a type system guaranteeing this.

The practical significance of uniform convergence is that *before* running SGD, for every error tolerance $\epsilon > 0$ we can find an accuracy coefficient $\eta > 0$ such that the difference between the smoothed and original objective function does not exceed ϵ, in particular for $\boldsymbol{\theta}^*$ delivered by the SGD run for the η-smoothed problem.

Discussion of Restrictions. To rule out the pathology of Example 6 we require that guards are non-0 almost everywhere.

Furthermore, as a consequence of the uniform limit theorem [29], (Unif) can only possibly hold if the *expectation* $\mathbb{E}_{\mathbf{s}\sim\mathcal{D}}\left[[\![M]\!](\boldsymbol{\theta}, \mathbf{s})\right]$ is continuous (as a function of the parameters $\boldsymbol{\theta}$). For a straightforward counterexample take $M \equiv \mathbf{if}\, \theta < 0\, \mathbf{then}\, \underline{0}\, \mathbf{else}\, \underline{1}$, we have $\mathbb{E}_{\mathbf{s}}[[\![M]\!](\theta)] = [\theta \geq 0]$ which is discontinuous, let alone differentiable, at $\theta = 0$. Our approach is to require that guards do not depend directly on parameters but they may do so, indirectly, via a diffeomorphic[8] reparameterisation transform; see Example 8. We call such guards *safe*.

In summary, our aim, intuitively, is to ensure that guards are the composition of a diffeomorphic transformation of the random samples (potentially depending on parameters) and a function which does not vanish almost everywhere.

5.1 Type System for Guard Safety

In order to enforce this requirement and to make the transformation more explicit, we introduce syntactic sugar, $\mathbf{transform\, sample}_{\mathcal{D}}\, \mathbf{by}\, T$, for applications of the form $T\, \mathbf{sample}_{\mathcal{D}}$.

Example 7. As expressed in Eq. (2), we can obtain samples from $\mathcal{N}(\mu, \sigma^2)$ via $\mathbf{transform\, sample}_{\mathcal{N}}\, \mathbf{by}\, (\lambda s.\, s \cdot \sigma + \mu)$, which is syntactic sugar for the term $(\lambda s.\, s \cdot \sigma + \mu)\, \mathbf{sample}_{\mathcal{N}}$.

We propose another instance of the generic type system of Section 4.3, $\vdash_{\text{unif}}$, where we annotate base types by $\alpha = (g, \Delta)$, where $g \in \{\mathbf{f}, \mathbf{t}\}$ denotes whether we seek to establish guard safety and Δ is a finite set of s_j capturing possible dependencies on samples. We subtype base types as follows: $\iota_1^{(g_1, \Delta_1)} \sqsubseteq_{\text{unif}} \iota_2^{(g_2, \Delta_2)}$

[8] [18, Example 12] illustrates why it is *not* sufficient to restrict the reparameterisation transform to *bijections* (rather, we require it to be a diffeomorphism).

if $\iota_1 \sqsubseteq \iota_2$ (as defined in Fig. 3a), $\Delta_1 \subseteq \Delta_2$ and $g_1 \preceq g_2$, where $\mathbf{t} \preceq \mathbf{f}$. This is motivated by the intuition that we can always drop[9] guard safety and add more dependencies.

The rule for conditionals ensures that only safe guards are used. The unary operations preserve variable dependencies and guard safety. Parameters and constants are not guard safe and depend on no samples (see [18] for the full type system):

$$\frac{\Gamma \mid \Sigma \vdash_{\text{unif}} L : \iota^{(\mathbf{t},\Delta)} \quad \Gamma \mid \Sigma' \vdash_{\text{unif}} M : \sigma \quad \Gamma \mid \Sigma'' \vdash_{\text{unif}} N : \sigma}{\Gamma \mid \Sigma +\!\!\!+ \Sigma' +\!\!\!+ \Sigma'' \vdash_{\text{unif}} \textbf{if}\, L < 0\, \textbf{then}\, M\, \textbf{else}\, N : \sigma}$$

$$\frac{}{\mid [] \vdash_{\text{unif}} \underline{-} : R^{(g,\Delta)} \to R^{(g,\Delta)}}$$

$$\frac{}{\theta_i : \iota^{(\mathbf{f},\emptyset)} \mid [] \vdash_{\text{unif}} \theta_i : \iota^{(\mathbf{f},\emptyset)}} \qquad \frac{}{\mid [] \vdash_{\text{unif}} \underline{r} : \iota^{(\mathbf{f},\emptyset)}}\; r \in [\![\iota]\!]$$

$$\frac{\boldsymbol{\theta} \mid [] \vdash_{\text{unif}} T : R^{\alpha} \to R^{\alpha}}{\boldsymbol{\theta} \mid [s_j \sim \mathcal{D}] \vdash_{\text{unif}} \textbf{transform}\, \textbf{sample}_{\mathcal{D}}\, \textbf{by}\, T : R^{(\mathbf{t},\{s_j\})}}\; T \text{ diffeomorphic}$$

A term $\boldsymbol{\theta} \mid [] \vdash_{\text{unif}} T : R^{\alpha} \to R^{\alpha}$ is diffeomorphic if $[\![T]\!](\boldsymbol{\theta}, []) = [\![T]\!]_{\eta}(\boldsymbol{\theta}, []) : \mathbb{R} \to \mathbb{R}$ is a diffeomorphism for each $\boldsymbol{\theta} \in \boldsymbol{\Theta}$, i.e. differentiable and bijective with differentiable inverse.

First, we can express affine transformations, in particular, the location-scale transformations as in Example 7:

Example 8 (Location-Scale Transformation). The term-in-context

$$\sigma : R^{(\mathbf{f},\emptyset)}_{>0}, \mu : R^{(\mathbf{f},\emptyset)} \mid [] \vdash \lambda s.\, \sigma \cdot s + \mu : R^{(\mathbf{f},\{s_1\})} \to R^{(\mathbf{f},\{s_1\})}$$

is diffeomorphic. (However for $\sigma : R^{(\mathbf{f},\emptyset)}$ it is *not* because it admits $\sigma = 0$.) Hence, the reparameterisation transform

$$G \equiv \sigma : R^{(\mathbf{f},\emptyset)}_{>0}, \mu : R^{(\mathbf{f},\emptyset)} \mid [s_1 : \mathcal{D}] \vdash \textbf{transform}\, \textbf{sample}_{\mathcal{D}}\, \textbf{by}\, (\lambda s. s \cdot \sigma + \mu) : R^{(\mathbf{t},\{s_1\})}$$

which has g-flag $\mathbf{t}$, is admissible as a guard term. Notice that G depends on the parameters, σ and μ, *indirectly* through a diffeomorphism, which is permitted by the type system.

If guard safety is sought to be established for the binary operations, we require that operands do not share dependencies on samples:

$$\frac{}{\mid [] \vdash_{\text{unif}} \underline{\circ} : \iota^{(\mathbf{f},\Delta)} \to \iota^{(\mathbf{f},\Delta)} \to \iota^{(\mathbf{f},\Delta)}}\; \circ \in \{+, \cdot\}$$

$$\frac{}{\mid [] \vdash_{\text{unif}} \underline{\circ} : \iota^{(\mathbf{t},\Delta_1)} \to \iota^{(\mathbf{t},\Delta_2)} \to \iota^{(\mathbf{t},\Delta_1 \cup \Delta_2)}}\; \circ \in \{+, \cdot\}, \Delta_1 \cap \Delta_2 = \emptyset$$

This is designed to address:

[9] as long as it is not used in guards

Example 9 (Non-Constant Guards). We have $\mid [] \vdash (\lambda x. x \underline{+} (\underline{-}x)) : R^{(\mathbf{f},\{s_1\})} \to R^{(\mathbf{f},\{s_1\})}$, noting that we must use $g = \mathbf{f}$ for the $\underline{+}$ rule; and because $R^{(\mathbf{t},\{s_j\})} \sqsubseteq_{\text{unif}} R^{(\mathbf{f},\{s_j\})}$, we have

$$\mid [] \vdash (\lambda x. x \underline{+} (\underline{-}x)) : R^{(\mathbf{t},\{s_1\})} \to R^{(\mathbf{f},\{s_1\})}.$$

Now $\textbf{transform}\,\textbf{sample}_{\mathcal{D}}\,\textbf{by}\,(\lambda y.y)$ has type $R^{(\mathbf{t},\{s_1\})}$ with the g-flag necessarily set to $\mathbf{t}$; and so the term

$$M \equiv \big(\lambda x. x \underline{+} (\underline{-}x)\big)\, \textbf{transform}\,\textbf{sample}_{\mathcal{D}}\,\textbf{by}\,(\lambda y.y)$$

which denotes 0, has type $R^{(\mathbf{f},\{s_1\})}$, but *not* $R^{(\mathbf{t},\{s_1\})}$. It follows that M cannot be used in guards (notice the side condition of the rule for conditional), which is as desired: recall Example 6. Similarly consider the term

$$N \equiv \big(\lambda x. (\lambda y\, z. \textbf{if}\; y \underline{+} (\underline{-}z) < 0\; \textbf{then}\, M_1\, \textbf{else}\, M_2)\, x\, x\big) \\ (\textbf{transform}\,\textbf{sample}_{\mathcal{D}}\,\textbf{by}\,(\lambda y.y)) \qquad (7)$$

When evaluated, the term $y \underline{+} (\underline{-}z)$ in the guard has denotation 0. For the same reason as above, the term N is not refinement typable.

The type system is however incomplete, in the sense that there are terms-in-context that satisfy the property (Unif) but which are not typable.

Example 10 (Incompleteness). The following term-in-context denotes the "identity":

$$\mid [] \vdash (\lambda x. (\underline{2} \cdot x) \underline{+} (\underline{-}x)) : R^{(\mathbf{t},\{s_1\})} \to R^{(\mathbf{f},\{s_1\})}$$

but it does *not* have type $R^{(\mathbf{t},\{s_1\})} \to R^{(\mathbf{t},\{s_1\})}$. Then, using the same reasoning as Example 9, the term

$$G \equiv (\lambda x. (\underline{2} \cdot x) \underline{+} (\underline{-}x))\, (\textbf{transform}\,\textbf{sample}_{\mathcal{D}}\,\textbf{by}\,(\lambda y.y))$$

has type $R^{(\mathbf{f},\{s_1\})}$, but *not* $R^{(\mathbf{t},\{s_1\})}$, and so $\textbf{if}\; G < 0\; \textbf{then}\, \underline{0}\, \textbf{else}\, \underline{1}$ is not typable, even though G can safely be used in guards.

5.2 Type Soundness

Henceforth, we fix parameters $\theta_1 : \iota_1^{(\mathbf{f},\emptyset)}, \ldots, \theta_m : \iota_m^{(\mathbf{f},\emptyset)}$.

Now, we address how to show property (Unif), i.e. that for $\boldsymbol{\theta} \mid \Sigma \vdash_{\text{unif}} M : \iota^{(g,\Delta)}$, the η-smoothed $\mathbb{E}[[\![M]\!]_\eta(\boldsymbol{\theta}, \mathbf{s})]$ converges uniformly for $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ as $\eta \searrow 0$. For this to hold we clearly need to require that σ_η has good (uniform) convergence properties (as far as the unavoidable discontinuity at 0 allows for):

Assumption 3 *For every $\delta > 0$, $\sigma_\eta \xrightarrow{unif.} [(-) > 0]$ on $(-\infty, -\delta) \cup (\delta, \infty)$.*

Observe that in general even if M is typable $[\![M]\!]_\eta$ does *not* converge uniformly in both $\boldsymbol{\theta}$ and $\mathbf{s}$ because $[\![M]\!]$ may still be discontinuous in $\mathbf{s}$:

Example 11. For $M \equiv \mathbf{if}\,(\mathbf{transform}\,\mathbf{sample}_{\mathcal{N}}\,\mathbf{by}\,(\lambda s.\, s + \theta)) < 0\,\mathbf{then}\,\underline{0}\,\mathbf{else}\,\underline{1}$, $[\![M]\!](\theta, s) = [s + \theta \geq 0]$, which is discontinuous, and $[\![M]\!]_\eta(\theta, s) = \sigma_\eta(s + \theta)$.

However, if $\boldsymbol{\theta} \mid \Sigma \vdash M : \iota^{(g,\Delta)}$ then $[\![M]\!]_\eta$ *does* converge to $[\![M]\!]$ uniformly almost uniformly, i.e., uniformly in $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ and *almost* uniformly in $\mathbf{s} \in \mathbb{R}^n$. Formally, we define:

Definition 4. *Let $f, f_\eta : \boldsymbol{\Theta} \times \mathbb{R}^n \to \mathbb{R}$, μ be a measure on $\mathbb{R}^n$. We say that f_η* converges uniformly almost uniformly *to f (notation: $f_\eta \xrightarrow{u.a.u.} f$) if there exist sequences $(\delta_k)_{k\in\mathbb{N}}$, $(\epsilon_k)_{k\in\mathbb{N}}$ and $(\eta_k)_{k\in\mathbb{N}}$ such that $\lim_{k\to\infty} \delta_k = 0 = \lim_{k\to\infty} \epsilon_k$; and for every $k \in \mathbb{N}$ and $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ there exists $U \subseteq \mathbb{R}^n$ such that*

1. *$\mu(U) < \delta_k$ and*
2. *for every $0 < \eta < \eta_k$ and $\mathbf{s} \in \mathbb{R}^n \setminus U$, $|f_\eta(\boldsymbol{\theta}, \mathbf{s}) - f(\boldsymbol{\theta}, \mathbf{s})| < \epsilon_k$.*

If f, f_η are independent of $\boldsymbol{\theta}$ this notion coincides with standard almost uniform convergence. For M from Example 11 $[\![M]\!]_\eta \xrightarrow{\text{u.a.u.}} [\![M]\!]$ holds although uniform convergence fails.

However, uniform almost uniform convergence entails uniform convergence of *expectations*:

Lemma 6. *Let $f, f_\eta : \boldsymbol{\Theta} \times \mathbb{R}^n \to \mathbb{R}$ have finite moments.*

If $f_\eta \xrightarrow{u.a.u.} f$ then $\mathbb{E}_{\mathbf{s}\sim\mathcal{D}}[f_\eta(\boldsymbol{\theta}, \mathbf{s})] \xrightarrow{unif.} \mathbb{E}_{\mathbf{s}\sim\mathcal{D}}[f(\boldsymbol{\theta}, \mathbf{s})]$.

As a consequence, it suffices to establish $[\![M]\!]_\eta \xrightarrow{\text{u.a.u.}} [\![M]\!]$. We achieve this by positing an infinitary logical relation between sequences of morphisms in **VectFr** (corresponding to the smoothings) and morphisms in **QBS** (corresponding to the measurable standard semantics). We then prove a fundamental lemma (details are in [18]). Not surprisingly the case for conditionals is most interesting. This makes use of Assumption 3 and exploits that guards, for which the typing rules assert the guard safety flag to be $\mathbf{t}$, can only be 0 at sets of measure 0. We conclude:

Theorem 1. *If $\theta_1 : \iota_1^{(\mathbf{f},\emptyset)}, \ldots, \theta_m : \iota_m^{(\mathbf{f},\emptyset)} \mid \Sigma \vdash_{\text{unif}} M : R^{(g,\Delta)}$ then $[\![M]\!]_\eta \xrightarrow{u.a.u.} [\![M]\!]$. In particular, if $[\![M]\!]_\eta$ and $[\![M]\!]$ also have finite moments then*

$$\mathbb{E}_{\mathbf{s}\sim\mathcal{D}}[[\![M]\!]_\eta(\boldsymbol{\theta}, \mathbf{s})] \xrightarrow{unif.} \mathbb{E}_{\mathbf{s}\sim\mathcal{D}}[[\![M]\!](\boldsymbol{\theta}, \mathbf{s})] \qquad \text{as } \eta \searrow 0 \text{ for } \boldsymbol{\theta} \in \boldsymbol{\Theta}$$

We finally note that $\vdash_{\text{unif}}$ can be made more permissible by adding syntactic sugar for a-fold (for $a \in \mathbb{N}_{>0}$) addition $\underline{a} \,\underline{\cdot}\, M \equiv M \,\underline{+}\, \cdots \,\underline{+}\, M$ and multiplication $M^{\underline{a}} \equiv M \,\underline{\cdot}\, \cdots \,\underline{\cdot}\, M$. This admits more terms as guards, but safely [18].

6 Related Work

[23] is both the starting point for our work and the most natural source for comparison. They correct the (biased) reparameterisation gradient estimator for non-differentiable models by additional non-trivial *boundary* terms. They present

an efficient method for *affine* guards only. Besides, they are not concerned with the *convergence* of gradient-based optimisation procedures; nor do they discuss how assumptions they make may be manifested in a programming language.

In the context of the reparameterisation gradient, [25] and [17] relax discrete random variables in a continuous way, effectively dealing with a specific class of discontinuous models. [39] use a similar smoothing for discontinuous optimisation but they do not consider a full programming language.

Motivated by guaranteeing absolute continuity (which is a necessary but not sufficient criterion for the correctness of e.g. variational inference), [24] use an approach similar to our trace types to track the samples which are drawn. They do not support standard conditionals but their "work-around" is also eager in the sense of combining the traces of both branches. Besides, they do not support a full higher-order language, in which higher-order terms can draw samples. Thus, they do not need to consider *function* types tracking the samples drawn during evaluation.

7 Empirical Evaluation

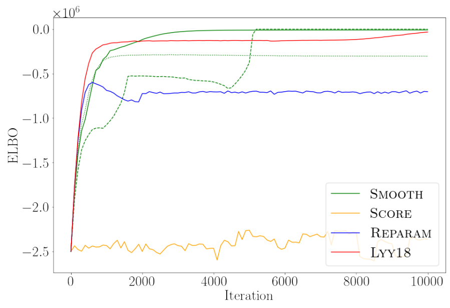
We evaluate our smoothed gradient estimator (Smooth) against the biased reparameterisation estimator (Reparam), the unbiased correction of it (LYY18) due to [23], and the unbiased (Score) estimator [31,38,27]. The experimental setup is based on that of [23]. The implementation is written in Python, using automatic differentiation (provided by the `jax` library) to implement each of the above estimators for an arbitrary probabilistic program. For each estimator and model, we used the Adam [19] optimiser for $10,000$ iterations using a learning rate of 0.001, with the exception of `xornet` for which we used 0.01. The initial model parameters $\boldsymbol{\theta}_0$ were fixed for each model across all runs. In each iteration, we used $N = 16$ Monte Carlo samples from the gradient estimator. For the Lyy18 estimator, a single subsample for the boundary term was used in each estimate. For our smoothed estimator we use accuracy coefficients $\eta \in \{0.1, 0.15, 0.2\}$. Further details are discussed in [18, Appendix E.1].

Compilation for First-Order Programs. All our benchmarks are first-order. We compile a potentially discontinuous program to a smooth program (parameterised by σ_η) using the compatible closure of

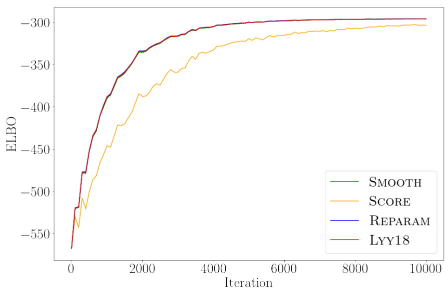
$$\mathbf{if}\, L < 0\, \mathbf{then}\, M\, \mathbf{else}\, N \rightsquigarrow (\lambda w.\, \sigma_\eta(-w) \cdot M + \sigma_\eta(w) \cdot N)\, L$$

Note that the size only increases linearly and that we avoid of an exponential blow-up by using abstractions rather than duplicating the guard L.

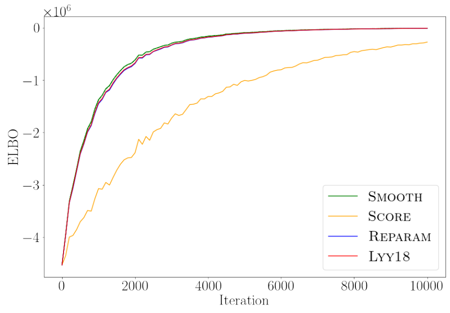
Models. We include the models from [23], an example from differential privacy [11] and a neural network for which our main competitor, the estimator of [23], is *not* applicable (see [18, Appendix E.2] for more details).



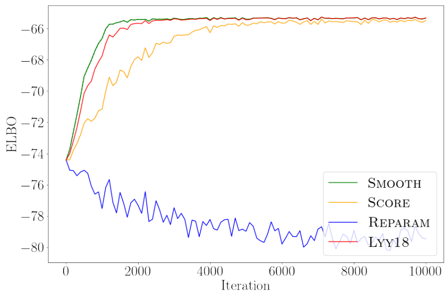
(a) `temperature`



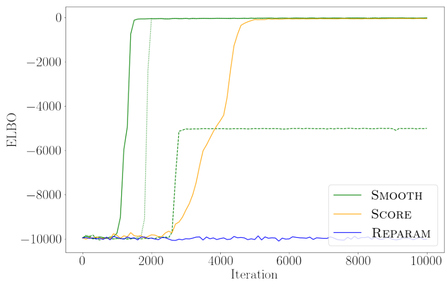
(b) `textmsg`



(c) `influenza`



(d) `cheating`



(e) `xornet`

Fig. 5: ELBO trajectories for each model. A single colour is used for each estimator and the accuracy coefficient $\eta = 0.1, 0.15, 0.2$ for Smooth is represented by dashed, solid and dotted lines respectively.

Analysis of Results

We plot the ELBO trajectories in Fig. 5 and include data on the computational cost and *work-normalised* variance [8] in [18, Table 2]. (Variances can be improved in a routine fashion by e.g. taking more samples.)

The ELBO graph for the `temperature` model in Fig. 5a and the `cheating` model in Fig. 5d shows that the Reparam estimator is biased, converging to suboptimal values when compared to the Smooth and Lyy18 estimators. For `temperature` we can also see from the graph and the data in [18, Table 2a] that the Score estimator exhibits extremely high variance, and does not converge.

Finally, the `xornet` model shows the difficulty of training step-function based neural nets. The Lyy18 estimator is not applicable here since there are non-affine conditionals. In Fig. 5e, the Reparam estimator makes no progress while other estimators manage to converge to close to 0 ELBO, showing that they learn a network that correctly classifies all points. In particular, the Smooth estimator converges the quickest.

Summa summarum, the results reveal where the Reparam estimator is biased and that the Smooth estimator does not have the same limitation. Where the Lyy18 estimator is defined, they converge to roughly the same objective value. Our smoothing approach is generalisable to more complex models such as neural networks with non-linear boundaries, as well as simpler and cheaper (there is no need to compute a correction term). Besides, our estimator has consistently significantly lower work-normalised variance, up to 3 orders of magnitude.

8 Conclusion and Future Directions

We have discussed a simple probabilistic programming language to formalise an optimisation problem arising e.g. in variational inference for probabilistic programming. We have endowed our language with a denotational (measurable) value semantics and a smoothed approximation of potentially discontinuous programs, which is parameterised by an accuracy coefficient. We have proposed type systems to guarantee pleasing properties in the context of the optimisation problem: For a fixed accuracy coefficient, stochastic gradient descent converges to stationary points even with the reparameterisation gradient (which is *unbiased*). Besides, the smoothed objective function converges uniformly to the true objective as the accuracy is improved.

Our type systems can be used to *independently* check these two properties to obtain partial theoretical guarantees even if one of the systems suffers from incompleteness. We also stress that SGD and the smoothed unbiased gradient estimator can even be applied to programs which are *not* typable.

Experiments with our prototype implementation confirm the benefits of reduced variance and unbiasedness. Compared to the unbiased correction of the reparameterised gradient estimator due to [23], our estimator has a similar convergence, but is simpler, faster, and attains orders of magnitude (2 to 3,000 x) reduction in work-normalised variance.

Future Directions. A natural avenue for future research is to make the language and type systems more complete, i.e. to support more well-behaved programs, in particular programs involving recursion.

Furthermore, the choice of accuracy coefficients leaves room for further investigations. We anticipate it could be fruitful not to fix an accuracy coefficient upfront but to gradually enhance it *during* the optimisation either via a predetermined schedule (dependent on structural properties of the program), or adaptively.

References

1. Aumann, R.J.: Borel structures for function spaces. Illinois Journal of Mathematics **5** (1961)
2. Bertsekas, D.: Convex optimization algorithms. Athena Scientific (2015)
3. Bertsekas, D.P., Tsitsiklis, J.N.: Gradient convergence in gradient methods with errors. SIAM J. Optim. **10**(3), 627–642 (2000)
4. Bingham, E., Chen, J.P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P.A., Horsfall, P., Goodman, N.D.: Pyro: Deep universal probabilistic programming. J. Mach. Learn. Res. **20**, 28:1–28:6 (2019)
5. Bishop, C.M.: Pattern recognition and machine learning, 5th Edition. Information science and statistics, Springer (2007)
6. Blei, D.M., Kucukelbir, A., McAuliffe, J.D.: Variational inference: A review for statisticians. Journal of the American Statistical Association **112**(518), 859–877 (2017)
7. Borgström, J., Lago, U.D., 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, ICFP 2016, Nara, Japan, September 18-22, 2016. pp. 33–46 (2016)
8. Botev, Z., Ridder, A.: Variance Reduction. In: Wiley StatsRef: Statistics Reference Online, pp. 1–6 (2017)
9. Cusumano-Towner, M.F., Saad, F.A., Lew, A.K., Mansinghka, V.K.: Gen: a general-purpose probabilistic programming system with programmable inference. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 221–236. ACM (2019)
10. Dahlqvist, F., Kozen, D.: Semantics of higher-order probabilistic programs with conditioning. Proc. ACM Program. Lang. **4**(POPL), 57:1–57:29 (2020)
11. Davidson-Pilon, C.: Bayesian Methods for Hackers: Probabilistic Programming and Bayesian Inference. Addison-Wesley Professional (2015)
12. Ehrhard, T., Tasson, C., Pagani, M.: Probabilistic coherence spaces are fully abstract for probabilistic PCF. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 309–320 (2014)
13. Frölicher, A., Kriegl, A.: Linear Spaces and Differentiation Theory. Interscience, J. Wiley and Son, New York (1988)
14. Heunen, C., Kammar, O., Staton, S., Yang, H.: A convenient category for higher-order probability theory. Proc. Symposium Logic in Computer Science (2017)
15. Heunen, C., Kammar, O., Staton, S., Yang, H.: A convenient category for higher-order probability theory. In: 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017. pp. 1–12 (2017)
16. Hur, C., Nori, A.V., Rajamani, S.K., Samuel, S.: A provably correct sampler for probabilistic programs. In: 35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India. pp. 475–488 (2015)
17. Jang, E., Gu, S., Poole, B.: Categorical reparameterization with gumbel-softmax. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings (2017)

18. Khajwal, B., Ong, C.L., Wagner, D.: Fast and correct gradient-based optimisation for probabilistic programming via smoothing (2023), https://arxiv.org/abs/2301.03415
19. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015)
20. Kingma, D.P., Welling, M.: Auto-encoding variational bayes. In: Bengio, Y., LeCun, Y. (eds.) 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings (2014)
21. Klenke, A.: Probability Theory: A Comprehensive Course. Universitext, Springer London (2014)
22. Lee, W., Yu, H., Rival, X., Yang, H.: Towards verified stochastic variational inference for probabilistic programs. PACMPL **4**(POPL) (2020)
23. Lee, W., Yu, H., Yang, H.: Reparameterization gradient for non-differentiable models. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada. pp. 5558–5568 (2018)
24. Lew, A.K., Cusumano-Towner, M.F., Sherman, B., Carbin, M., Mansinghka, V.K.: Trace types and denotational semantics for sound programmable inference in probabilistic languages. Proc. ACM Program. Lang. **4**(POPL), 19:1–19:32 (2020)
25. Maddison, C.J., Mnih, A., Teh, Y.W.: The concrete distribution: A continuous relaxation of discrete random variables. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings (2017)
26. Mak, C., Ong, C.L., Paquet, H., Wagner, D.: Densities of almost surely terminating probabilistic programs are differentiable almost everywhere. In: Yoshida, N. (ed.) Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12648, pp. 432–461. Springer (2021)
27. Minh, A., Gregor, K.: Neural variational inference and learning in belief networks. In: Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014. JMLR Workshop and Conference Proceedings, vol. 32, pp. 1791–1799. JMLR.org (2014)
28. Mohamed, S., Rosca, M., Figurnov, M., Mnih, A.: Monte carlo gradient estimation in machine learning. J. Mach. Learn. Res. **21**, 132:1–132:62 (2020)
29. Munkres, J.R.: Topology. Prentice Hall, New Delhi,, 2nd. edn. (1999)
30. Murphy, K.P.: Machine Learning: A Probabilististic Perspective. MIT Press (2012)
31. Ranganath, R., Gerrish, S., Blei, D.M.: Black box variational inference. In: Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014. pp. 814–822 (2014)
32. Rezende, D.J., Mohamed, S., Wierstra, D.: Stochastic backpropagation and approximate inference in deep generative models. In: Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014. JMLR Workshop and Conference Proceedings, vol. 32, pp. 1278–1286. JMLR.org (2014)
33. Stacey, A.: Comparative smootheology. Theory and Applications of Categories **25**(4), 64–117 (2011)

34. Staton, S.: Commutative semantics for probabilistic programming. In: Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. pp. 855–879 (2017)
35. Staton, S., Yang, H., Wood, F.D., Heunen, C., Kammar, O.: Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016. pp. 525–534 (2016)
36. Titsias, M.K., Lázaro-Gredilla, M.: Doubly stochastic variational bayes for non-conjugate inference. In: Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014. pp. 1971–1979 (2014)
37. Vákár, M., Kammar, O., Staton, S.: A domain theory for statistical probabilistic programming. PACMPL **3**(POPL), 36:1–36:29 (2019)
38. Wingate, D., Weber, T.: Automated variational inference in probabilistic programming. CoRR **abs/1301.1299** (2013)
39. Zang, I.: Discontinuous optimization by smoothing. Mathematics of Operations Research **6**(1), 140–152 (1981)
40. Zhang, C., Butepage, J., Kjellstrom, H., Mandt, S.: Advances in Variational Inference. IEEE Trans. Pattern Anal. Mach. Intell. **41**(8), 2008–2026 (2019)

Type-safe Quantum Programming in Idris

Liliane-Joy Dandy[1,2,3], Emmanuel Jeandel[3], and Vladimir Zamdzhiev[3,4](✉)

[1] EPFL, Lausanne, Switzerland
liliane-joy.dandy@epfl.ch
[2] École polytechnique, Palaiseau, France
[3] Université de Lorraine, CNRS, Inria, LORIA, 54000 Nancy, France
emmanuel.jeandel@loria.fr
[4] Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190, Gif-sur-Yvette, France
vladimir.zamdzhiev@inria.fr

Abstract. Variational Quantum Algorithms are hybrid classical-quantum algorithms where classical and quantum computation work in tandem to solve computational problems. These algorithms create interesting challenges for the design of suitable programming languages. In this paper we introduce Qimaera, which is a set of libraries for the Idris 2 programming language that enable the programmer to implement hybrid classical-quantum algorithms where the full power of the elegant Idris language works in synchrony with quantum programming primitives. The two key ingredients of Idris that make this possible are (1) dependent types which allow us to implement unitary quantum operations; and (2) linearity which allows us to enforce fine-grained control over the execution of quantum operations so that we may detect and reject many physically inadmissible programs. We also show that Qimaera is suitable for variational quantum programming by providing implementations of two prominent variational quantum algorithms – QAOA and VQE.

1 Introduction

Variational Quantum Algorithms [30,25,13] present a computational paradigm where hybrid classical-quantum algorithms work in tandem to solve computational problems. The classical part of the algorithm is performed by a classical processor and the quantum part of the algorithm is executed on a quantum device. During the computation process, intermediary results produced by the quantum device are passed onto the classical device which performs further computation on them that is used to tune the parameters of the quantum part of the algorithm, which therefore has an effect on the quantum dynamics. The hybrid classical-quantum back and forth process repeats until a desired termination condition is satisfied.

This hybrid classical-quantum computational paradigm opens up interesting and important challenges for the design of suitable programming languages. It is clear that if we wish to program within such computational scenarios, we

Source code for Qimaera [1] and a full version of the paper [12] are available.

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 507–534, 2023.
https://doi.org/10.1007/978-3-031-30044-8_19

need to develop a language that correctly models the manipulation of *quantum resources*. In particular, quantum measurements give rise to *probabilistic computational effects* that are inherited by the classical side of the language. Another issue is that quantum information behaves very differently compared to classical information. As an example, quantum information cannot be copied in a uniform way [36], unlike classical information, which may be freely copied without restriction. Therefore, if we wish to avoid runtime errors, the quantum fragment of the language needs to be equipped with features for fine-grained control, such as for example, having a *substructural typing discipline* [16,8,7,24,6] where contraction (i.e., copying) is restricted. On the other hand, when doing classical computation, such restrictions are unnecessary and often inconvenient. One solution to this problem is to design a language with a classical (non-linear) fragment together with a quantum (linear) one, both of which interact nicely with each other. In fact, this can be achieved within an existing language that has a sufficiently advanced type system, as we show in this paper.

In this paper, we describe *Qimaera* (named after the hybrid creature Chimaera from Greek mythology), which is a set of libraries for the Idris 2 language [10] that allow the programmer to implement hybrid quantum-classical algorithms in a *type-safe* way. Idris 2 is an elegant functional programming language that is equipped with an advanced type system based on Quantitative Type Theory [24,6] that brings many useful features to the programmer, most notably *dependent types* and *linearity*. These two features of Idris are crucial for the development of Qimaera and, in fact, are the reason we chose Idris in the first place. Dependent types are used throughout our entire development in order to correctly represent and formalise the compositional nature of quantum operations. Linearity is used in order to enforce the proper consumption of quantum resources (during execution) in a way that is admissible with respect to the laws of quantum mechanics. The combination of dependent types and linearity allows us to *statically* detect and reject erroneous quantum programs and this ensures the type safety of our approach to variational quantum programming.

In our intended computational scenario, we have access to both a classical computer and a quantum computer. Since we cannot directly observe quantum information, we directly interact with the classical computer which sends instructions to, and receives data from, the quantum device via a suitable interface that makes use of the IO monad. In our view, this is a representation of a (perhaps simple) computational environment for hybrid quantum-classical programming. We design a suitable (abstract) interface that allows us to model this situation accurately and which makes use of the IO monad. However, since the authors do not personally have any quantum hardware, we provide only one concrete implementation of our interface that simulates the relevant quantum operations on our classical computers by using the proper linear-algebraic formalism, but while still using the IO monad as prescribed by the abstract interface. From a high-level programming perspective, the abstract interface addresses the programming challenges induced by the classical-quantum device scenario, but it ignores lower-level considerations (e.g., error correction).

We emphasise that we can achieve type-safe hybrid quantum-classical programming in an *existing* programming language by implementing suitable libraries. This is important for *variational* quantum programming, because in most variational quantum algorithms, the classical part of the algorithm is considerably larger, more complicated and more difficult to implement, compared to the quantum part of the algorithm. Therefore, it is important for the programming language to have first-class support for classical programming features. We think our chosen language, Idris, is such a language. The advanced type system of Idris allows us to elegantly mix quantum and classical programming primitives and therefore allows us to achieve our objectives. We demonstrate that Qimaera is suitable for variational quantum programming by providing implementations of the two most prominent variational quantum algorithms – QAOA and VQE. Moreover, our implementation of these algorithms has been achieved in a *type-safe* programming framework. By this we mean that common quantum programming errors (copying of qubits, applying a CNOT operation with the same source and target, etc.) are *statically* detected and rejected by the Idris type checker. We also note that being able to combine quantum and classical programming is important in other scenarios too (for instance in quantum cryptography).

Quantum Circuits vs Recursive Quantum Programs. We want to stress that the focus of our paper is not about quantum circuits, but about (recursive) quantum *programs and algorithms*. While some quantum algorithms may be seen as quantum circuits, there are algorithms which are more general, for example, repeat-until-success (see §5.2) and variational quantum algorithms (see §6). Such algorithms are not quantum circuits in the traditional understanding of this notion, and for them general recursion, probabilistic effects and classical computation might be important.

More specifically, general recursion is important, because many existing quantum algorithms are *probabilistic* and find the correct answer with some probability. General recursion then allows the programmer to repeatedly run such an algorithm until the correct solution is found, thereby resulting in an almost-surely-terminating program, i.e., a program that terminates with probability 1. However, since there is no upper bound on the number of runs of the algorithm, general recursion is necessary to express this pattern. For instance, this can be used to repeatedly run Shor's algorithm until the algorithm succeeds in finding a divisor. This might also be useful for variational quantum algorithms, because it allows us to express more flexible termination conditions, which give us more than simple iterations.

Safety Properties. We consider type safety in quantum programming to be important, because it is easy to make mistakes where one can copy qubits or forget to use a qubit. The former is physically inadmissible due to the *no-cloning theorem* of quantum mechanics [36] and the latter usually leads to unexpected behaviour, because discarding quantum information causes a *side effect* that may affect the rest of the quantum system. These observations suggest that we may design our systems and libraries carefully, by utilising *linear typing features*, so that these situations can be *statically* detected and rejected by the type system, therefore avoiding the problem. Otherwise, such situations could result in *runtime errors* (e.g., copying a qubit), which are clearly undesirable. In fact, in our experience, it is very easy to make such mistakes and this happened while we were implementing some of the quantum algorithms described in this paper. Our type-safe approach to quantum programming automatically detects and rejects these kinds of erroneous programs during type checking. While we do not have any proof of correctness, we believe that our approach is type-safe as long as the users do not modify our library files.

Why Idris instead of another language? The features that we require to achieve our objectives are: general recursion, dependent types and linearity. We chose Idris 2, because it is an excellent language that has all three of these features. Removing general recursion limits the expressivity of the language (as explained above). The other two features are used to reject erroneous quantum programs. We think that most programming languages that have the three features mentioned above are suitable for type-safe hybrid quantum-classical programming. In fact, one of the main points that we wish to demonstrate with this paper is that it is *not* necessary to build a standalone programming language in order to achieve the desired safety properties. Instead, the same can be achieved with already *existing* languages, such as Idris 2. This approach has some advantages (compared to designing a standalone language), such as: easier maintenance, larger library support, better integration with the newest developments in classical programming, etc.

2 Background on Quantum Computation

Readers interested in a detailed introduction to quantum computing may consult [26]. In this section we summarise the basic notions that are relevant for our development.

The simplest non-trivial quantum system is the *quantum bit*, often abbreviated as *qubit*. Qubits may be thought of as the quantum counterparts of the bit from classical computation. A qubit $|\psi\rangle$ is represented as a normalised vector in $\mathbb{C}^2$. The *computational basis* is given by the pair of vectors $|0\rangle \overset{\text{def}}{=} \begin{pmatrix}1\\0\end{pmatrix}$ and $|1\rangle \overset{\text{def}}{=} \begin{pmatrix}0\\1\end{pmatrix}$, which may be seen as representing the classical bits 0 and 1. An arbitrary qubit is described by $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$ where $a, b \in \mathbb{C}$ and $|a|^2 + |b|^2 = 1$.

A qubit may be in (uncountably) many different states, whereas a classical bit is either 0 or 1. When the linear combination $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$ is non-trivial, then we say that $|\psi\rangle$ is in *superposition* of $|0\rangle$ and $|1\rangle$. Superposition is a very important quantum resource which is used by many quantum algorithms.

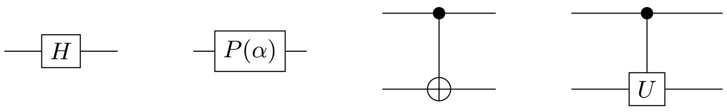


Fig. 1. The Hadamard, Phase Shift, CNOT and CU gates.

The state space that describes a system of n qubits is the Hilbert space $\mathbb{C}^{2^n}$. If $|\psi\rangle$ and $|\phi\rangle$ are two states of n and m qubits respectively, then the composite $n+m$ qubit state $|\psi\phi\rangle \overset{\text{def}}{=} |\psi\rangle \otimes |\phi\rangle$ is described by the Kronecker product $\otimes$ of the original states.

A quantum state $|\psi\rangle \in \mathbb{C}^{2^n}$ may undergo a *unitary evolution* described by a unitary matrix $U \in \mathbb{C}^{2^n \times 2^n}$ in which case the new state of the system is described by the vector $U|\psi\rangle$. Unitary operations (and matrices) are closed under sequential composition (described by matrix multiplication $\circ$) and under parallel composition (described by Kronecker product $\otimes$). Sequential composition of unitary operations is used to describe the temporal evolution of quantum systems, whereas the parallel composition is used to describe their spatial structure.

The unitary quantum operations are also often called *unitary gates*. One typically chooses a *universal gate set* which is a small set of unitary operations that suffices to express all other unitary operations via (parallel and sequential) composition. The universal gate set that we choose for our development is standard and we specify these unitary operations next by giving their action on the computational basis (which uniquely determines the operations).

The *Hadamard Gate*, denoted H, is the 1-qubit unitary map whose action on the computational basis is given by $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)$ and $H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)$ and its primary purpose is to generate superposition. The *Phase Shift Gate*, denoted $P(\alpha)$, for $\alpha \in \mathbb{R}$, is a 1-qubit unitary map whose action on the computational basis is given by: $P(\alpha)|0\rangle = |0\rangle$ and $P(\alpha)|1\rangle = e^{i\alpha}|1\rangle$ and its primary purpose is to modify the phase of a quantum state. The family of Phase Shift Gates is parameterised by the choice of $\alpha \in \mathbb{R}$ and important special cases include the unitary gates $T \overset{\text{def}}{=} P(\pi/4)$ and $Z \overset{\text{def}}{=} P(\pi)$. The *Controlled-Not Gate* (CNOT), is a 2-qubit unitary map whose action on the computational basis is given by $\text{CNOT}|00\rangle = |00\rangle$; $\text{CNOT}|01\rangle = |01\rangle$; $\text{CNOT}|10\rangle = |11\rangle$ and $\text{CNOT}|11\rangle = |10\rangle$ and this unitary map may be used to generate quantum entanglement.

Unitary gates admit a diagrammatic representation as *quantum circuits*. The atomic unitary gates we described above are shown in Figure 1. Composite unitary gates may also be described as circuits (see Figure 2): sequential composition amounts to plugging wires of subdiagrams and parallel composition amounts to juxtaposition.

The CNOT gate is the simplest example of a *controlled unitary gate*. Given a unitary gate $U : \mathbb{C}^{2^n} \to \mathbb{C}^{2^n}$, the controlled-$U$ unitary gate is the unitary gate $CU : \mathbb{C}^{2^{n+1}} \to \mathbb{C}^{2^{n+1}}$ whose action is determined by the assignments $CU(|0\rangle \otimes |\psi\rangle) = |0\rangle \otimes |\psi\rangle$ and $CU(|1\rangle \otimes |\psi\rangle) = |1\rangle \otimes (U|\psi\rangle)$. Controlled unitary operations are ubiquitous in quantum computing (see Figure 1 for their circuit depiction).

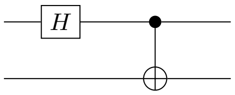


Fig. 2. A quantum circuit that may be used for the preparation of the Bell state.

Every unitary operation U is *reversible* with the inverse operation given by the conjugate transpose, denoted $U^\dagger$, which is again a unitary matrix. Applying the inverse operation (i.e., the *adjoint*) of a given unitary map is ubiquitous.

A quantum state $|\psi\rangle \in \mathbb{C}^{2^n}$, with $n > 1$, is said to be *entangled* when there exists no non-trivial decomposition $|\psi\rangle = |\phi\rangle \otimes |\tau\rangle$. Quantum entanglement is a very important resource in quantum computation which is exhibited by many quantum algorithms. Because of the possibility of entanglement, we cannot, in general, break down quantum systems into smaller components and we are often forced to reason about such systems in their entirety. A very important example of an entangled state is the *Bell state* given by $|\mathrm{Bell}\rangle \overset{\mathrm{def}}{=} \frac{|00\rangle+|11\rangle}{\sqrt{2}}$.

Preparing a new qubit in state $|0\rangle$ is an admissible physical operation. This, together with application of unitary gates as part of the computation, allows us to prepare arbitrary quantum states, e.g., the Bell state can be prepared by taking $|\mathrm{Bell}\rangle = (\mathrm{CNOT} \circ (H \otimes I))\,|00\rangle$ (see Figure 2).

Quantum information cannot be directly observed without affecting the state of the underlying system. In order to extract information from quantum systems, we need to perform a *quantum measurement* on (parts of) our systems. For example, when performing a quantum measurement on a qubit in the state $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$, there are two possible outcomes: either the quantum system will collapse to state $|0\rangle$ and we obtain the classical bit 0 as evidence of this event, or, the quantum system will collapse to state $|1\rangle$ and we obtain the classical bit 1 as evidence of this event. The first outcome (corresponding to bit 0) occurs with probability $|a|^2$ and the second outcome (corresponding to bit 1) occurs with probability $1 - |a|^2 = |b|^2$. In general, when we measure n qubits simultaneously, we obtain a bit string of length n which determines the event that occurred and the quantum system collapses to a corresponding state with some probability, both of which are determined via the Born rule of quantum mechanics. Therefore, quantum measurements induce evolutions which are *probabilistic* and *irreversible* (or *destructive*), which distinguishes them from unitary evolutions, which are *deterministic* and *reversible*.

Unlike classical information, quantum information cannot be uniformly copied. This is made precise by the *no-cloning* theorem [36]. There exists no unitary operation $U : \mathbb{C}^4 \to \mathbb{C}^4$, such that for every qubit $|\psi\rangle : U(|\psi\rangle \otimes |0\rangle) = |\psi\rangle \otimes |\psi\rangle$. This means that copying of quantum information is a *physically inadmissible* operation. Ideally, quantum programming languages should be designed so that these kinds of errors are detected during type checking.

3 Background on the Idris 2 Language

In this section, we give a short overview of the Idris 2 language and its main features that are relevant for the development of Qimaera. Idris 2 is a functional language with a syntax influenced by that of Haskell. The features of particular interest for us are dependent types and linearity, both of which are crucial for Qimaera. Its type system is based on Quantitative Type Theory [24,6], which specifies how dependent types and linearity are combined.

Dependent Types. In Idris, types are first-class primitives and they may be manipulated like other constructs of the language. This allows us to formulate more expressive types that can depend on values, and hence it enables us to make some properties and program invariants explicit.

Example 1. The type of vectors is a simple and useful example of a dependent type. A vector is a list with a fixed length that is part of the type. It can be defined as follows, where `S` is the successor function for natural numbers, and `a` is a polymorphic type:

```
data Vect : Nat -> Type -> Type where
  Nil  : Vect 0 a
  (::) : a -> Vect k a -> Vect (S k) a
```

The type `Vect` has two constructors (i.e., introduction rules). The first one constructs the empty vector, of length zero. The second one is used to introduce non-empty vectors: a vector with `k+1` elements of type `a` is constructed by combining an element of type `a` and a vector of size `k`.

Type dependency allows us to specify useful program properties and type checking ensures that they hold. For instance, we can define an `append` function that concatenates two vectors. Then, the size of the output vector is the sum of the sizes of the input vectors and this is specified by its type.

```
append : Vect n a -> Vect m a -> Vect (n + m) a
```

This information allows the language to detect a larger class of programming errors. Note that type dependency information is not available for the analogous function on lists. Type dependency may also be used to express constraints on the inputs of a function, e.g., we can define a *total* function, called `pop`, that cannot be applied to an empty vector.

```
pop : Vect (S k) a -> Vect k a
pop (x :: xs) = xs
```

Writing “`pop []`” is now an error which is detected statically, rather than dynamically, and we note that the same cannot be achieved if we were to replace vectors with lists.

Linearity. The type system of Idris 2 is based on Quantitative Type Theory, where every function argument is associated with a multiplicity that states the number of times the variable is used at runtime[5]. This multiplicity can be 0, 1 or ω. An argument with multiplicity 0 is only used at compile time (to determine type dependency information) and is erased at runtime. A *linear* argument has multiplicity 1 and it is used exactly once at runtime. Finally, ω represents the unrestricted multiplicity, which is default, where the function argument may be used any number of times.

Example 2. Consider the `pop` function which we just discussed. The (implicitly bound) variables `k` and `a` have multiplicity 0, because they are not explicitly specified as separate arguments, and they are *not* accessible at runtime in the function. The variables `x` and `xs`, which are explicitly bound, have the default (unrestricted) multiplicity.

Example 3. An important type which we define in Qimaera is the type of linear vectors, which we write as `LVect`. The only difference, compared to the standard vectors in Idris, is that the `(::)` constructor for `LVect` is a linear function in all of its arguments. Linearity in Idris 2 is specified by writing the multiplicity 1 in front of each argument.

```
data LVect : Nat -> Type -> Type where
  Nil : LVect 0 a
  (::) : (1 _ : a) -> (1 _ : LVect k a) ->
         LVect (S k) a
```

We also use linear pairs that are already defined in Idris 2.

```
data LPair : Type -> Type -> Type
  (#) : (1 _ : a) -> (1 _ : b) -> LPair a b
```

Linearity allows us to specify and enforce constraints on function arguments, e.g., it prevents us from duplicating data, so the function definition below leads to an error:

```
copy : (1 _ : a) -> LPair a a
copy x = x # x

Error: While processing right hand side of
copy. There are 2 uses of linear name x.
```

Linearity is prominently used in Qimaera. In particular, when manipulating quantum data, linearity is enforced in order to properly handle quantum resources and comply with the laws of quantum mechanics.

Remark 1. We learned only recently that there is a type of linear vectors in the Idris libraries. In the future we might replace our implementation with the one provided by the Idris developers.

[5] This can be understood similarly to how variables are used in linear λ-calculi.

```
data Unitary : Nat -> Type where
  IdGate : Unitary n
  H      : (j : Nat) ->
           {auto prf : (j < n) = True} ->
           Unitary n -> Unitary n
  P      : (p : Double) -> (j : Nat) ->
           {auto prf : (j < n) = True} ->
           Unitary n -> Unitary n
  CNOT   : (c : Nat) -> (t : Nat) ->
           {auto prf1 : (c < n) = True} ->
           {auto prf2 : (t < n) = True} ->
           {auto prf3 : (c /= t) = True} ->
           Unitary n -> Unitary n
```

Fig. 3. The Unitary data type (file: `Unitary.idr`).

4 Unitary Operations in Qimaera

We describe our representation of unitary transformations in Qimaera as an algebraic data type called `Unitary`. Every value of this type is, by design, an *algebraic decomposition* of a unitary operation in terms of the atomic unitary gates that we selected in §2.

The `Unitary` data type allows us to adopt a high-level *algebraic* and *scalable* approach towards the reversible fragment of quantum computation. This provides the programmer with some benefits as we show in this section. However, using the `Unitary` data type is actually entirely optional. Users who are interested in effectful quantum programming do *not* have to use it (see §5) and they may still do hybrid classical-quantum programming, but at the cost of losing the algebraic decomposition of unitary operations. However, there are many useful functions that are available for manipulating values of type `Unitary` that are not available for effectful quantum programs.

4.1 The Unitary Data Type

Quantum unitary operations admit an algebraic representation based on the atomic gates from the universal gate set we described. Our idea for the representation of unitary operations is based on this, or equivalently, on how unitary operations may be expressed in terms of unitary quantum circuit diagrams. Because of these reasons, linearity is not required for our formalisation of unitary operations. The code for the `Unitary` data type is listed in Figure 3 and we now describe our representation in greater detail.

Given a natural number `n : Nat`, the type of unitary operations on `n` qubits is given by `Unitary n`. Note that `Unitary` is an algebraic data type with a simple type dependency on the arity of the desired operation. The `Unitary` type has four different introduction rules which we describe next.

The first constructor, `IdGate`, represents the identity unitary operation on `n` qubits. Diagramatically, we can see this as constructing a circuit of `n` wires, without applying any other gates on any of the wires. It has a unique argument, `n`, which is implicit – it can be omitted when calling the `IdGate` constructor and it will often be inferred by Idris.

The second constructor, `H`, should be understood as applying the Hadamard gate H to the `j`-th qubit of some previously constructed unitary circuit which is specified as the last argument. The first implicit argument, `n`, is simply the arity of the resulting unitary operation. The second implicit argument, `prf`, is a proof obligation that `j` is smaller than `n`. This ensures that the argument `j` identifies an existing wire of the previously constructed unitary circuit (last argument) and therefore the overall definition is algebraically and physically sound. We think that the implicit argument `prf` may be removed from our implementation if we change the type of `j` to `Fin n`, the type of natural numbers less than `n`. However, in our experience, we found it easier to work with the current implementation rather than with `Fin` and for this reason we chose to keep the `prf` argument.

The third constructor, `P`, should be viewed as applying the $P(p)$ gate, where the real number $p \in \mathbb{R}$ is approximated by the term `p : Double`.[6] The remaining arguments serve the same purpose as those for `H`.

The final constructor, `CNOT`, should be understood as applying the CNOT gate, where `c` identifies the wire used for the control (the small black dot in Figure 1), `t` identifies the wire of the target (the crossed circle in Figure 1) and the last (unnamed) argument is the previously constructed unitary circuit on which we are applying CNOT. The remaining arguments are implicit: the argument `n` is the arity of the unitary; `prf1` and `prf2` ensure that `c` and `t` identify valid wires of the unitary circuit; `prf3` ensures that the control and target wires are *distinct* and therefore the overall application of CNOT is physically and algebraically admissible.

In our representation of quantum unitary operations, we make use of type dependency to impose proof obligations on some of our constructors in order to guarantee that the representation makes sense in physical and algebraic terms. Indeed, this might sometimes be a burden for the users of the library. However, Idris can sometimes automatically infer the required proofs without any assistance from the user, e.g., when all arguments are statically known constants (see Example 4). This is discussed in detail in the next subsection.

4.2 Constructing Unitary Transformations

The four basic introduction rules of the `Unitary` type allow us to define *high-level functions* in Idris that can be used to construct complex unitary circuits out of simpler ones. We discuss this here and we show that the proof obligations

[6] This approximation is not a big limitation – in fault-tolerant quantum computing one usually replaces the $P(p)$ gate family with a single $T = P(\pi/4)$ gate and the resulting gate set suffices to achieve approximation with *arbitrary* precision. So we can easily replace `P` with a `T` constructor.

from Figure 3 can sometimes be ameliorated and sometimes even completely sidestepped.

First, we point out that auto-implicit arguments may occasionally be inferred by Idris via suitable search. For example, if all the arguments are known statically, the required proofs will often be discovered by Idris and then the users do not have to manually provide them.

Example 4. The unitary circuit from Figure 2 may be constructed in the following way:

```
toBellBasis : Unitary 2
toBellBasis = CNOT 0 1 (H 0 IdGate)
```

In this example, Idris is able to infer all the implicit arguments and there is no need to provide any proofs. If we do not satisfy one of the constraints, e.g., if we write `CNOT 1 1` above (which does not make physical sense), then we get the following error during type checking:

```
Error : While processing right hand side of
toBellBasis. Can't find an implementation for
not (== 1 1) = True.
```

An error also is reported if we provide a wire number larger than 1. It also is useful to define *standalone* unitary gates for the $H, P(r)$ and CNOT gates as follows:

```
HGate : Unitary 1
HGate = H 0 IdGate

PGate : Double -> Unitary 1
PGate r = P r 0 IdGate

CNOTGate : Unitary 2
CNOTGate = CNOT 0 1 IdGate
```

Composing Unitary Circuits. Our libraries provide functions for sequential composition (`compose`) and parallel composition (`tensor`) of unitary operations:

```
compose : Unitary n -> Unitary n -> Unitary n
tensor : {n : Nat} -> {p : Nat} -> Unitary n
                   -> Unitary p -> Unitary (n + p)
```

Notice that both functions do not require proof obligations like the ones from Figure 3. This means that one of the main algebraic ways for composing unitary operations may be done without requiring such proofs. The use of these functions is ubiquitous in practice and we introduce the infix synonyms `(.)` and `(#)` for `compose` and `tensor`, respectively.

Example 5. The `toBellBasis` gate from Example 4 may be equivalently expressed in the following way:

```
toBellBasis : Unitary 2
toBellBasis = CNOTGate . (HGate # IdGate)
```

Qimaera provides another, more general, form of composition via the function `apply` whose type is as follows:

```
apply : {i : Nat} -> {n : Nat} ->
        Unitary i -> Unitary n ->
        (v : Vect i Nat) ->
        {auto _ : isInjective n v = True} ->
        Unitary n
```

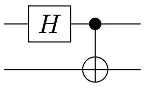
The `apply` function is used to apply a smaller unitary circuit of size `i` to a bigger one of size `n`, giving the vector `v` of wire indices on which we wish to apply the smaller circuit. It needs one auto-implicit proof which enforces the consistency requirement that all indices of the wires specified by `v` are pairwise distinct and smaller than `n`. In fact, the `apply` function implements the most general notion of composition that we support. Both sequential and parallel composition can be realised as special cases using it. The importance of the vector `v` is that it determines how to apply the smaller unitary circuit of arity `i` to *any selection* of `i` wires of the larger unitary circuit, and moreover, it also allows us to *permute* the inputs/outputs of the smaller unitary circuit while doing so. More specifically, if the k-th entry of the vector `v` is the natural number p, then the k-th input/output of the smaller unitary circuit will be applied to the p-th wire of the larger unitary circuit. This is best understood by example.

Example 6. Consider the following code sample:

```
U : Unitary 3
U = HGate # IdGate {n = 1} # (PGate pi)

apply_example : Unitary 3
apply_example = apply toBellBasis U v
```

where `v` is a vector of length two. Here, `toBellBasis` is given in Example 4 and represents the circuit given below left; `U` represents the circuit given below right:



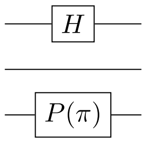


Table 1 shows what unitary circuit is specified under different values of `v`. In these cases, Idris can automatically infer the required proofs and the user does not have to provide them.

Remark 2. Instead of using `apply`, there is another possible approach, in the spirit of *symmetric monoidal categories* [23, §XI], where we could add one extra introduction rule to the `Unitary` type for representing *permutations* of wires. However, in our view, this approach is less appealing, because one does not usually think of permutations (induced by the symmetric monoidal structure) as physical gates.

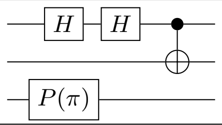
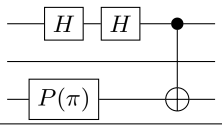
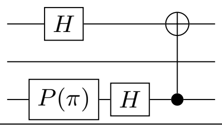
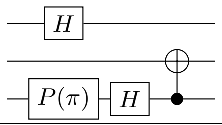
`apply toBellBasis U [0,1]`	
`apply toBellBasis U [0,2]`	
`apply toBellBasis U [2,0]`	
`apply toBellBasis U [2,1]`	

Table 1. Examples illustrating the `apply` function.

Adjoints of Unitary Circuits. Qimaera also provides a function

```
adjoint : Unitary n -> Unitary n
```

which computes the adjoint (i.e., inverse) of a given unitary circuit. One often has to apply the inverse of a given unitary circuit, so having a method such as this one is useful. Our implementation uses the standard approach for synthesising the adjoint. The adjoint may be used, for example, to uncompute the result of the application of unitary gates on auxiliary qubits.

Controlled Unitary Circuits. We also implement a function

```
controlled : {n : Nat} -> Unitary n -> Unitary (S n)
```

which given a unitary circuit U constructs the corresponding controlled unitary circuit CU. Our implementation uses the standard and simple algorithm for doing this, but more efficient algorithms may also be implemented in principle.

Analysis of Unitary Circuits. Unitary circuits are represented in a scalable way in Qimaera and we can use Idris to optimise them. In particular, the function:

```
optimise : Unitary n -> Unitary n
```

may be used to optimise a given unitary circuit by reducing the number of gates while keeping the action of the circuit unchanged. So far, this function provides only very basic optimisations, but more sophisticated and powerful ones may be added in principle. The point we wish to make is that unitary circuits in Qimaera may be analysed and manipulated like other algebraic data

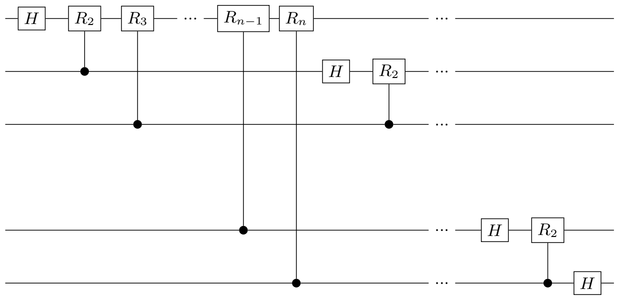


Fig. 4. The QFT unitary circuit on n qubits.

type structures using the capabilities of Idris. In fact, the file `Unitary.idr` also provides other functions that do this. For example, we provide functions for calculating the circuit depth, calculating the number of specific atomic gates used by a circuit, drawing circuits in the terminal and exporting circuits to Qiskit so that users may then use external analysis tools.

4.3 Example: The Quantum Fourier Transform

The Quantum Fourier Transform (QFT) is an important unitary operator that is used in Shor's polynomial-time algorithm for integer factorisation [34]. The unitary circuit which realises QFT on n qubits is shown in Figure 4, where $R_n \overset{\text{def}}{=} P\left(\frac{2\pi}{2^n}\right)$. The Qimaera code which implements this unitary circuit is shown in Figure 5. Notice that we make use of the `controlled` function from §4.2 in the function `cRm`, so that we can implement the controlled R_n gates that are required. In this example, we have parameters that are universally quantified, so we need a few proofs in the code: one for using the `apply` function and one for correctly unifying the size of the circuit. These proof obligations appear when writing the `qftRec` function and Idris did not infer them automatically, so we had to provide the proofs. To get some intuition for the code: the `qftRec` function computes the recursive pattern that applies a Hadamard gate followed by the cascade of controlled R_n gates; the `qft` function then computes the other recursive pattern which consists in repeatedly using the pattern computed by `qftRec` and composing as appropriate.

5 Effectful Quantum Computation

In the previous section we showed how unitary circuits can be represented in Qimaera. This suffices to capture the pure, deterministic and reversible fragment of quantum computation. However, we need to also consider effectful and probabilistic quantum processes which may result from quantum measurements, because this is important for hybrid quantum-classical computation. In this section, we show how this can be done in a type-safe way by using monads, linearity and dependent types.

```
Rm : Nat -> Unitary 1
Rm m = PGate (2 * pi / (pow 2 (cast m)))

cRm : Nat -> Unitary 2
cRm m = controlled (Rm m)

qftRec : (n : Nat) -> Unitary n
qftRec 0 = IdGate
qftRec 1 = HGate
qftRec (S (S k)) =
  let t =  (qftRec (S k)) # IdGate
  in rewrite sym $ lemmaplusOneRight k
  in apply (cRm (S (S k))) t [S k,0]
           {prf = lemmaInj1 k}

qft : (n : Nat) -> Unitary n
qft 0 = IdGate
qft (S k) =
  let g = qftRec (S k)
      h = (IdGate {n = 1}) # (qft k)
  in h . g
```

Fig. 5. Qimaera code for QFT (file: `QFT.idr`).

5.1 Representation of Quantum Effects in Qimaera

We now explain how the quantum program dynamics are represented in Qimaera in a type-safe way. We are (roughly) inspired by representing the notion of a *quantum configuration* as it appears in [32,29,22], which is in turn used to formally describe the operational semantics of quantum type systems.

Qubits in Qimaera. Because of the possibility of quantum entanglement, we cannot describe the state of an individual qubit which is part of a larger composite system. On the other hand, we wish to be able to refer to *parts* of the whole system by identifying specific qubit positions. In Qimaera, we introduce the following type declaration:

```
data Qubit : Type where
  MkQubit : (n : Nat) -> Qubit
```

The argument of type `Nat` is used as a *unique identifier* for the constructed qubit. The constructor `MkQubit` is *private* and users of our libraries cannot access it (outside of the library file). Instead, our libraries provide functions (Figure 7) that ensure that a term of type `Qubit` is created with a fresh (i.e., unique) natural number that serves as its identifier within a monadic environment. This is handled by our functions through careful manipulation of the available data within the monadic environment. In fact, these functions are the expected way for our users to access or manipulate qubits and, moreover, our users cannot access the unique identifiers (unless they modify our libraries). This allows us to formulate a representation where values of type `Qubit` unambiguously refer to the relevant parts of larger composite systems. Therefore, a value of type `Qubit` should be understood as a pointer, or as a unique identifier, of a 1-qubit subsystem of some larger quantum state. Terms of type `Qubit` do not carry any sort of linear-algebraic information.

Probabilistic Effects. Quantum measurements induce probabilistic computational effects which are inherited by the classical side of the computation in hybrid classical-quantum algorithms. Furthermore, in our intended computational scenario, the classical computer (on which Idris is running) sends instructions to, and receives data from, the quantum device. In order to correctly model all of this, it is clear that we have to use the IO monad in order to encapsulate these effects. However, when representing quantum program dynamics, we also need to *enforce linearity*, but all the functions provided by the IO monad (e.g., `pure` which introduces pure values to monadic types) are *not* linear in any of their arguments. This creates a problem which may be solved by using the LIO library, which extends the IO monad with linearity. For brevity, we define `R` to be our linear IO monad:

```
R : Type -> Type
R = L IO {use = Linear}
```

Then, by using `R` we can combine IO effects (and thus also probabilistic effects) and linearity in a suitable way.

Quantum State Transformer. Quantum computation is *effectful*, and moreover, quantum information *cannot* be observed by the classical computer (on which Idris is running): it only receives classical information through communication with the quantum device. Because of this, we adopt a more abstract view on the hybrid classical-quantum computational process. In order to do this, we define an (abstract) *quantum state transformer* by combining several different concepts: *indexed state monads* [4][7], linearity and IO (and thus also probabilistic) effects. Our representation of these ideas in Qimaera is shown in Figure 6, where we omit the function definitions for brevity.

The type `QStateT` is parameterised by a choice of three (arbitrary) types, so it is fairly abstract. Soon, we will see that it is very useful for our purposes. The intended interpretation of this type is the following: any value of type

```
QStateT initialType finalType returnType
```

represents a stateful (quantum) computation starting from a (quantum) state of type `initialType` and ending in a (quantum) state of type `finalType` which

[7] See [33] for a Haskell implementation of this idea.

```
data QStateT :  Type -> Type -> Type -> Type where
  MkQST : (1 _ : (1 _ : initialType) ->
          R (LPair finalType returnType)) ->
          QStateT initialType finalType returnType

  runQStateT : (1 _ : initialType) ->
               (1 _ : QStateT initialType finalType returnType) ->
               R (LPair finalType returnType)

  pure : (1 _ : a) -> QStateT t t a

  (>>=) : (1 _ : QStateT i m a) ->
          (1 _ : ((1 _ : a) -> QStateT m o b)) ->
          QStateT i o b
```

Fig. 6. Quantum state transformer (file: `QStateT.idr`).

produces a user-accessible result of type `returnType` during the computation. For example, a value of type

```
QStateT (LPair Qubit Qubit) Qubit Bool
```

should be understood as a quantum process that transforms a two-qubit state into a single-qubit state and returns a single (classical) value of type `Bool` to the user. The functions presented in Figure 6 allow us to adopt a *monadic programming discipline* when working with `QStateT` and we do so henceforth. We remark that `QStateT` makes use of the monad `R` which encapsulates the IO (and probabilistic) effects and that linearity is enforced when working with `QStateT`.

Effectful Quantum Programming. The `QStateT` monad can be used to define a suitable *abstract interface* for quantum programming. In Figure 7, we present an excerpt of the `QuantumOp` interface which allows us to write quantum programs and execute them in a type-safe way. All of the hybrid quantum-classical algorithms we present are implemented using this interface.

The function `newQubits` is used to prepare `p` new qubits in state $|0\rangle$ and the function returns a linear vector of length `p` with the qubit identifiers of the newly created qubits. The function `applyUnitary` is used to apply a unitary operation of arity `i` to the qubits specified by the argument `LVect` (which also determines the order of application) and the operation returns an `LVect` which serves the same purpose – it identifies the qubits which were just modified by the unitary operator. The file `QuantumOp.idr` also provides functions `applyH`, `applyP` and `applyCNOT` which can be seen as special cases of `applyUnitary`. However, these three functions do not depend on the `Unitary` type.

The `measure` function is used to measure `i` qubits identified by the `LVect` argument and it returns a value of type `Vect i Bool` that represents the result of the measurement. After this, the `i` measured qubits are not reused, as one can see from the provided type information.

```
interface QuantumOp (0 t : Nat -> Type) where
  newQubits : (p : Nat) -> QStateT (t n) (t (n+p)) (LVect p Qubit)

  newQubit : QStateT (t n) (t (S n)) Qubit

  applyUnitary : {n : Nat} -> {i : Nat} -> (1 _ : LVect i Qubit) ->
    Unitary i -> QStateT (t n) (t n) (LVect i Qubit)

  applyH : {n : Nat} -> (1 _ : Qubit) -> QStateT (t n) (t n) Qubit

  applyP : {n : Nat} -> Double -> (1 _ : Qubit) ->
                        QStateT (t n) (t n) Qubit

  applyCNOT : {n : Nat} -> (1 _ : Qubit) -> (1 _ : Qubit) ->
    QStateT (t n) (t n) (LPair Qubit Qubit)

  measure : {n : Nat} -> {i : Nat} -> (1 _ : LVect i Qubit) ->
    QStateT (t (i + n)) (t n) (Vect i Bool)

  measureQubit : {n : Nat} -> (1 _ : Qubit) ->
                              QStateT (t (S n)) (t n) Bool

  measureAll : {n : Nat} -> (1 _ : LVect n Qubit) ->
    QStateT (t n) (t 0) (Vect n Bool)

  run : QStateT (t 0) (t 0) (Vect n Bool) -> IO (Vect n Bool)
```

Fig. 7. The `QuantumOp` interface (file: `QuantumOp.idr`).

Finally, the function `run` is used to *execute* quantum algorithms on the quantum device and obtain the classical information returned from it. Notice that `run` can be used to execute effectful quantum processes which start from the trivial quantum state (on zero qubits) and which terminate in the same trivial quantum state, but which also produce some number of classical bits as a user-accessible return result. This may be used to run quantum algorithms: in a typical situation, we start with the trivial quantum state (on zero qubits), we prepare n qubits in state $|0\rangle$, we apply some unitary operations on them, and we finally measure all the qubits, thereby producing n bits of classical information. This quantum algorithm may then be represented as a value of type `QStateT (t 0) (t 0) (Vect n Bool)`. Running it, however, produces a classical value of type `IO (Vect n Bool)`, because the execution is probabilistic and because our classical computer (on which we are running Idris) has to perform IO actions to communicate with the quantum device.

In fact, *all* of the above operations modify the quantum state on the quantum device and may cause IO effects, because of the need to communicate with the quantum device. This is indeed reflected by our interface. Observe, that our interface is defined using the `QStateT` monad transformer which does incorporate IO effects (via the `R` monad we discussed previously).

Example 7. A fair coin toss may be implemented using quantum resources. The process is simple: (1) prepare the state $|0\rangle$; (2) apply the H gate to it; (3) measure the qubit and return this as output. We implement this as follows:

```
coin : QuantumOp t => IO Bool
coin = do
  [b] <- run (do
          q <- newQubit {t = t}
          q <- applyH q
          r <- measure [q]
          pure r
        )
  pure b
```

The top-level **do** block simply realises monadic sequencing for the standard IO monad. The **do** block within the `run` environment is more interesting and crucial for our development. It performs monadic sequencing for the `QStateT` monad and it represents the simple three-step algorithm we just described. The call to the `run` function executes this algorithm and users obtain the produced classical information by storing it in the variable `b` of type `Bool`. We emphasise that linearity is *enforced* within the `run` environment and this is what brings safety properties in our approach, e.g., all of the following scenarios are statically detected and rejected by Idris: passing the qubit `q` to a non-linear function, copying the qubit `q`, forgetting to measure the qubit `q`. For example, if in the above code we replace the last two statements in the `run` environment with "`pure True`", then Idris statically detects this error.

The function `coin` from Example 7 is implemented using our *abstract* interface. This means we can use this function in any *concrete* implementation of the `QuantumOp` interface. Since the authors do not have any quantum hardware, we provide one concrete implementation of this interface, called `SimulatedOp`, which performs linear-algebraic simulation of all the required operations. For example, if we wish to use the `coin` function, then the code:

```
testCoin : IO Bool
testCoin = coin {t = SimulatedOp}
```

defines a new function, called `testCoin`, which does the same as `coin`, but it specifically instructs Idris to use linear-algebraic simulation. We emphasise that all of our quantum algorithms are written using our abstract interface, so there is no need to reimplement them for any additional concrete implementations of the interface.

5.2 Example: Repeat-Until-Success Algorithm

Repeat-until-success (RUS) [27] is an algorithm for implementing quantum unitary operators by using *quantum measurements* and *general unbounded recursion*. The main advantage in using RUS over traditional deterministic techniques

```
RUS : QuantumOp t => (1 _ : Qubit) ->
      (u' : Unitary 2) -> (e : Unitary 1) ->
      QStateT (t 1) (t 1) Qubit
RUS q u' e = do
  q' <- newQubit
  [q',q] <- applyUnitary [q',q] u'
  b <- measureQubit q'
  if b then do
         [q] <- applyUnitary [q] (adjoint e)
         RUS q u' e
       else pure q

example_u' : Unitary 2
example_u' = H 0 $ T 0 $ CNOT 0 1 $ H 0 $ CNOT 0 1 $ T 0 $
             H 0 IdGate

runRUS : QuantumOp t => IO Bool
runRUS = do
  [b] <- run (do
               q <- newQubit {t = t}
               q <- RUS q example_u' IdGate
               measure [q]
         )
  pure b

testRUS : IO Bool
testRUS = runRUS {t = SimulatedOp}
```

Fig. 8. Repeat-until-success algorithm (file: `RUS.idr`).

that synthesise unitary operators, is that with RUS the expected number of T gates (which are expensive in terms of error correction[8]) can be reduced.

In the simplest case, we wish to realise a fixed single-qubit unitary operator $U : \mathbb{C}^2 \to \mathbb{C}^2$. The RUS algorithm is as follows. Given an input qubit $|\psi\rangle$, then: (1) prepare a new qubit in state $|0\rangle$; (2) apply a two-qubit unitary operator U' (chosen in advance depending on U); (3) measure the first qubit; (4) if the measurement outcome is 0 (which occurs with probability $p > 0$), then the output state is $U|\psi\rangle$, as required, and the algorithm terminates; otherwise the current state is $E|\psi\rangle$, where E is some other unitary operator (chosen in advance depending on U), so we apply $E^\dagger$ to this state and we go back to step (1). The unitary operators U' and E are chosen in advance, depending on U, before the algorithm starts so that the above conditions are satisfied. Note that synthesising U' and E is not part of the algorithm and we do not discuss this here.

Assuming that appropriate U' and E are chosen, this process always terminates in state $U|\psi\rangle$ (provided $p > 0$) so RUS indeed implements the unitary operator U. Note that this is an *algorithmic* realisation of U, not an algebraic one, and so we cannot write a program of type `Unitary` that achieves this. Instead, we represent this as a quantum program in Figure 8. There, `RUS q u'`

[8] We do not automatically implement error correction, so it has to be handled either by the developer or provided by the quantum device on the remote end.

`e` is the quantum state transformer which implements the RUS algorithm as above. The function `runRUS` simply executes the RUS algorithm on a qubit in state $|0\rangle$, with the unitary operator chosen from [27, Figure 8], then measures the qubit and returns the outcome. Both of these functions are written using our abstract interface. The function `testRUS` is the same as `runRUS`, but it also instructs Idris to use linear-algebraic simulation for the execution. Note that, in our implementation, we have taken a specific instance of RUS by choosing U' to be the unitary operator described by `example_u'` as discussed in [27, Figure 8].

Remark 3. The `run(-)` environment enforces linearity, so if we wish to use the `RUS` function within it, then the qubit argument must be linear in `RUS`.

6 Variational Quantum Programming

In the previous section we saw that Qimaera is suitable for writing recursive and effectful quantum programs that make use of quantum measurements. Moreover, Idris 2 is an excellent programming language with an advanced type system and first-class support for classical programming features. In order to demonstrate that Qimaera is suitable for hybrid classical-quantum programming, we also have to show that both classical and quantum programming features may be elegantly combined. This is the purpose of this section and we achieve this by implementing the two most prominent variational quantum algorithms: the Quantum Approximate Optimization Algorithm (QAOA) [13] and the Variational Quantum Eigensolver (VQE) [30]. In this paper we only describe QAOA. See the full paper [12] for more information on the implementation of VQE.

The objective of QAOA is to try to find the minimum (or maximum) eigenvalue of a Hamiltonian. A Hamiltonian is a Hermitian (i.e., self-adjoint) matrix $\mathcal{H}$ (we use a calligraphic font to differentiate it from H, the Hadamard matrix). Its minimum eigenvalue is the minimum (real) value λ such that $\mathcal{H}|\psi\rangle = \lambda|\psi\rangle$ for some nonzero vector $|\psi\rangle$. As $\mathcal{H}$ is unitarily diagonalizable, this is equivalent to the minimum of $\langle\psi|\mathcal{H}|\psi\rangle$ for all vectors $|\psi\rangle$ of norm 1, where $\langle\psi| \stackrel{\text{def}}{=} |\psi\rangle^\dagger$.

QAOA starts with some assumption on what the vector $|\psi\rangle$ looks like and usually $|\psi\rangle$ is prepared by a quantum circuit that depends on some real parameters $\alpha_1, \ldots, \alpha_p$. By measuring this state $|\psi\rangle$, one obtains some information on the value of $\langle\psi|\mathcal{H}|\psi\rangle$. This information can then be fed to a *classical* optimizer to change the value of the parameters $\alpha_1, \ldots, \alpha_p$ for subsequent execution.

This classical-quantum back and forth is repeated until some satisfactory termination condition has been satisfied. For example, we may simply repeat this process k times, where $k \in \mathbb{N}$ is some constant, but more sophisticated termination conditions are also possible. However, there is no guarantee that we will find the minimum eigenvalue.

Implementation of QAOA. QAOA is a variational algorithm [13] that approximately solves optimization problems. Let $f : \{0,1\}^n \to \mathbb{R}$ be a function for which we want to find its minimum. We see f as a diagonal Hamiltonian over n

qubits defined by $\mathcal{H}\left|x\right\rangle = f(x)\left|x\right\rangle$ for all $x \in \{0,1\}^n$. We are therefore searching for the minimum eigenvalue of this Hamiltonian.

In this case, the state $|\psi\rangle$ that minimises the Hamiltonian $\mathcal{H}$ is often assumed to be of the form: $|\psi\rangle = (HP(\beta_p)H)^{\otimes n}e^{\gamma_p \mathcal{H}} \cdots (HP(\beta_1)H)^{\otimes n}e^{\gamma_1 \mathcal{H}}H^{\otimes n}\left|0\right\rangle$. The depth parameter $p \in \mathbb{N}$ is usually fixed to be small, and we have a guarantee that the results of our algorithm become better when p becomes larger. To be able to produce a circuit which computes $|\psi\rangle$, the Hamiltonian $\mathcal{H}$ may be assumed to have a special form so that we can make a circuit for $e^{\gamma\mathcal{H}}$. A well-known and important example is to compute the maximum cut of an undirected graph, i.e., to solve the MAXCUT problem.

Our implementation for QAOA on the MAXCUT problem is presented in the file `QAOA.idr` and an excerpt is shown in Figure 9. The problem depends on the graph G for which we want the maximum cut, a depth parameter p, and some real parameters β_i, γ_i.

In our implementation, we have a function `QAOA_Unitary`, that takes these parameters as input and produces a unitary circuit that may be used to prepare the state $|\psi\rangle$ when applied to the initial state $\left|0\right\rangle^{\otimes n}$. We then measure this state $|\psi\rangle$ and present the result (a cut of the graph in the obvious binary encoding) to an optimiser. Our optimiser is implemented by the function `classicalOptimisation` that uses all observable information from all previous runs (which amounts to the values of the parameters β_i, γ_i and the value of the cuts that have been previously obtained through quantum measurements) to compute the subsequent rotation parameters β_i, γ_i that we will use for the next iteration. The type of this function indicates that it uses the IO monad: this is because we wish to allow the function to use probabilistic optimisation algorithms or even external tools. One of the simplest implementations of this function chooses the rotation parameters at random.

The interplay between the classical and the quantum part is presented in Figure 9. The function `QAOA` takes as input a natural number k representing how many times the whole routine will be done, the depth p of the circuit, and the graph G on which to compute the cut. Notice that the call to the quantum device is isolated inside the `run` function.

7 Related Work

In this section we compare Qimaera with other existing quantum programming languages that are implemented in software. We omit comparisons with quantum type systems that do not have a software implementation. We provide a feature comparison with some quantum programming languages in Table 2 and we now clarify the meaning of some of the selected features.

By *Type Safety* we mean that the language can statically detect (and reject) erroneous programs which duplicate quantum data. *General Recursion* is the ability to express recursive (possibly non-terminating) programs and almost-surely-terminating programs, such as RUS (see §5.2). *Measurements* is the ability to use the outcomes of quantum measurements in the control flow of programs.

```
QAOA_Unitary : {n : Nat} -> (betas : Vect p Double)
                         -> (gammas : Vect p Double)
                         -> (graph: Graph n) -> Unitary n

classicalOptimisation : {p : Nat}
                     -> (graph : Graph n)
                     -> (previous_info : Vect k (Vect p Double,
                        Vect p Double, Cut n))
                     -> IO (Vect p Double, Vect p Double)

QAOA' : QuantumOp t =>
        {n : Nat} ->
        (k : Nat) -> (p : Nat) -> (graph : Graph n) ->
        IO (Vect k (Vect p Double, Vect p Double, Cut n))
QAOA' 0 p graph = pure []
QAOA' (S k) p graph = do
  previous_info <- QAOA' {t} k p graph
  (betas, gammas) <- classicalOptimisation graph previous_info
  let circuit = QAOA_Unitary betas gammas graph
  cut <- run (do
             qs <- newQubits {t} n
             qs <- applyUnitary qs circuit
             measureAll qs
             )
  pure $ (betas, gammas, cut) :: previous_info

QAOA : QuantumOp t => {n : Nat} -> (k : Nat) -> (p : Nat) ->
                                   Graph n -> IO (Cut n)
QAOA k p graph = do
  res <- QAOA' {t} k p graph
  let cuts = map (\(_, _, cut) => cut) res
  let (cut,size) = bestCut graph cuts
  pure cut
```

Fig. 9. Qimaera implementation (excerpt) for the QAOA algorithm solving the MAX-CUT problem.

Promotion of Measurements is the ability to integrate the outcomes of quantum measurements as a *native* classical type (e.g., `Bool`): this essentially allows us to switch from a quantum mode of operation into a classical one and allows us to use both quantum and classical programming paradigms; it may be roughly understood as corresponding to the *promotion* rule of linear logic [16]. For *Higher-order Functions* we distinguish between purely classical ones and mixed classical-quantum (in the second column); some languages support both, but treat the quantum ones non-linearly which may cause loss of type safety. Finally, by *Effects* we mean the ability to incorporate probabilistic computational effects (which are an essential part of the dynamics of hybrid classical-quantum programs) and also IO (input/output) effects into our programming workflow.

The QWIRE language [28,31] and the SQIR language [20,19] are quantum circuit languages that are embedded in the Coq proof assistant [11]. Both of these languages have access to dependent types, courtesy of Coq. The focus of these languages is mostly on verification, whereas in Qimaera we focus on *programming* and Idris 2 has better support for classical, quantum and effectful programming features compared to Coq. Both QWIRE and SQIR represent quantum primitives through the use of low-level specification languages that are embedded in Coq: both of these specification languages lack the ability to express quantum algorithms that require general recursion and both of them lack the ability to express quantum higher-order functions. Because of the former reason, the RUS algorithm from §5.2 cannot be expressed in QWIRE or SQIR.

Silq [9] is a standalone quantum programming language which also is type-safe and whose main notable feature is automatic uncomputation of temporary values. We currently partially support this feature, because we have clearly identified and separated the reversible fragment of quantum computation (see the `Unitary` type) and we can synthesise the required adjoints by calling the `adjoint` function. Compared to Silq, the main advantage of Qimaera is that Idris has better support for classical programming features and so we believe that Qimaera is a better choice for hybrid classical-quantum programming. In addition, Silq does not support general recursion, so it cannot express quantum algorithms that rely on this (e.g., RUS §5.2).

Language	Type Safety	General Recursion	Dependent Types	Measurements	Promotion of Measurements	Higher-order Functions		Effects
						Classical	Quantum	
Quipper	✕	✓	✕	✓	✓	✓	(non-linear)	✓
Proto-Quipper-D	✓	✓	✓	✕	✕	✓	✓	✕
Proto-Quipper-Dyn	✓	✓	✕	✓	✓	✓	✓	✓
QWIRE	✓	✕	✓	✓	✕	✓	✕	✕
SQIR	✓	✕	✓	✓	✕	✓	✕	✕
Silq	✓	✕	(limitted)	✓	✓	✓	✓	✕
Qiskit	✕	✓	✕	✓	✓	✓	(non-linear)	✓
Q#	✕	✓	✕	✓	✓	✓	(non-linear)	✓
Cirq	✕	✓	✕	✓	✓	✓	(non-linear)	✓
Qimaera	✓	✓	✓	✓	✓	✓	✓	✓

Table 2. Feature comparison between Qimaera and other languages.

Quipper [18] and the Quantum IO monad (QIO) [3] are two domain specific languages (DSLs) embedded in Haskell. Neither of them are type safe because they do not utilise linearity and they cannot statically detect quantum programs that are physically inadmissible. However, thanks to the language similarities between Haskell and Idris, the programming style in these languages is somewhat similar to ours (e.g., all three use monads). In our view, both of these papers have been influential for the design of functional quantum programming languages.

Another recent language includes Proto-Quipper-D [14] which is a type-safe circuit description language. This language is based on a novel type system which shows how linearity and dependent types can be combined. A fundamental difference between Proto-Quipper-D and Qimaera is that linearity is the default mode of operation in Proto-Quipper-D, whereas in Qimaera the default mode is non-linear. The focus in Proto-Quipper-D is on *circuit* description and generation and the language currently lacks effectful quantum measurements and probabilistic effects, so it cannot be used for variational quantum programming at present. Another related language is Proto-Quipper-Dyn [15]. It is similar to Proto-Quipper-D, but it lacks dependent types (which Qimaera has). On the other hand, it can handle quantum measurements and has *dynamic lifting*, i.e., the ability to parameterize quantum circuits based on information observed from quantum measurements. Note that Qimaera also has dynamic lifting.

Other languages, include Google's Cirq [17] (a set of python libraries), IBM's Qiskit [2] (a set of python libraries) and Microsoft's Q$\sharp$ [35] (standalone). These languages offer a wide-range of quantum functions and features, however, none of them are type-safe. Qimaera does not have this problem and this is indeed its main advantage over them, together with dependent types.

8 Future Work

For future work, it would be interesting to consider methods that would allow us to reduce some of the proof obligations that are imposed by the `Unitary` data type. Going beyond Idris and our library, another natural direction is to consider whether programming languages that support substructural approaches other than linearity (e.g., uniqueness types, ownership) can be used to achieve type-safe quantum programming. It would also be interesting to consider the relevance of arrows [21,5] in quantum programming. Furthermore, implementing and testing our abstract interface on an actual hybrid quantum-classical hardware environment would most likely bring additional challenges.

Acknowledgements. We thank Robert Rand for discussions about this paper. We also thank the anonymous referees for their feedback which lead to multiple improvements of this paper. EJ is supported by the PEPR integrated project EPiQ and the European Project NEASQC (Grant Agreement 951821). Most of the work was done in LORIA/Inria Nancy during an Inria internship of the first author who was a student at École polytechnique. The first and last authors have changed affiliations since then.

References

1. Qimaera github repository. https://github.com/zamdzhiev/Qimaera, accessed: 30.01.2023
2. Aleksandrowicz, G., Alexander, T., Barkoutsos, P., Bello, L., Ben-Haim, Y., Bucher, D., Cabrera-Hernández, F.J., Carballo-Franquis, J., Chen, A., Chen, C.F., Chow, J.M., Córcoles-Gonzales, A.D., Cross, A.J., Cross, A., Cruz-Benito, J., Culver, C., González, S.D.L.P., Torre, E.D.L., Ding, D., Dumitrescu, E., Duran, I., Eendebak, P., Everitt, M., Sertage, I.F., Frisch, A., Fuhrer, A., Gambetta, J., Gago, B.G., Gomez-Mosquera, J., Greenberg, D., Hamamura, I., Havlicek, V., Hellmers, J., Łukasz Herok, Horii, H., Hu, S., Imamichi, T., Itoko, T., Javadi-Abhari, A., Kanazawa, N., Karazeev, A., Krsulich, K., Liu, P., Luh, Y., Maeng, Y., Marques, M., Martín-Fernández, F.J., McClure, D.T., McKay, D., Meesala, S., Mezzacapo, A., Moll, N., Rodríguez, D.M., Nannicini, G., Nation, P., Ollitrault, P., O'Riordan, L.J., Paik, H., Pérez, J., Phan, A., Pistoia, M., Prutyanov, V., Reuter, M., Rice, J., Davila, A.R., Rudy, R.H.P., Ryu, M., Sathaye, N., Schnabel, C., Schoute, E., Setia, K., Shi, Y., Silva, A., Siraichi, Y., Sivarajah, S., Smolin, J.A., Soeken, M., Takahashi, H., Tavernelli, I., Taylor, C., Taylour, P., Trabing, K., Treinish, M., Turner, W., Vogt-Lee, D., Vuillot, C., Wildstrom, J.A., Wilson, J., Winston, E., Wood, C., Wood, S., Wörner, S., Akhalwaya, I.Y., Zoufal, C.: Qiskit: An open-source framework for quantum computing (Jan 2019). https://doi.org/10.5281/zenodo.2562111, https://doi.org/10.5281/zenodo.2562111
3. Altenkirch, T., Green, A.S.: The quantum IO monad. Semantic Techniques in Quantum Computation pp. 173–205 (2010)
4. Atkey, R.: Parameterised notions of computation. J. Funct. Program. **19**(3-4), 335–376 (2009). https://doi.org/10.1017/S095679680900728X, https://doi.org/10.1017/S095679680900728X
5. Atkey, R.: What is a categorical model of arrows? Electron. Notes Theor. Comput. Sci. **229**(5), 19–37 (2011). https://doi.org/10.1016/j.entcs.2011.02.014, https://doi.org/10.1016/j.entcs.2011.02.014
6. Atkey, R.: Syntax and semantics of quantitative type theory. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 56–65. ACM (2018). https://doi.org/10.1145/3209108.3209189, https://doi.org/10.1145/3209108.3209189
7. Benton, P.N., Wadler, P.: Linear logic, monads and the lambda calculus. In: LICS 1996 (1996)
8. Benton, P.: A mixed linear and non-linear logic: Proofs, terms and models. In: Computer Science Logic: 8th Workshop, CSL '94, Selected Papaers (1995). https://doi.org/10.1007/BFb0022251, http://dx.doi.org/10.1007/BFb0022251
9. Bichsel, B., Baader, M., Gehr, T., Vechev, M.T.: Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 286–300. ACM (2020). https://doi.org/10.1145/3385412.3386007, https://doi.org/10.1145/3385412.3386007
10. Brady, E.C.: Idris 2: Quantitative type theory in practice. In: Møller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference). LIPIcs, vol. 194, pp. 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik

(2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.9, https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

11. Coq Development Team: The Coq proof assistant reference manual. https://coq.inria.fr/distrib/current/refman/ (2021), accessed: 19.11.2021
12. Dandy, L.J., Jeandel, E., Zamdzhiev, V.: Type-safe quantum programming in Idris. https://doi.org/10.48550/ARXIV.2111.10867, https://arxiv.org/abs/2111.10867
13. Farhi, E., Goldstone, J., Gutmann, S.: A quantum approximate optimization algorithm (2014)
14. Fu, P., Kishida, K., Ross, N.J., Selinger, P.: A tutorial introduction to quantum circuit programming in dependently typed Proto-Quipper. In: Lanese, I., Rawski, M. (eds.) Reversible Computation - 12th International Conference, RC 2020, Oslo, Norway, July 9-10, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12227, pp. 153–168. Springer (2020). https://doi.org/10.1007/978-3-030-52482-1_9, https://doi.org/10.1007/978-3-030-52482-1_9
15. Fu, P., Kishida, K., Ross, N.J., Selinger, P.: Proto-Quipper with dynamic lifting. CoRR **abs/2204.13041** (2022). https://doi.org/10.48550/arXiv.2204.13041, https://doi.org/10.48550/arXiv.2204.13041
16. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**, 1 – 101 (1987)
17. Google AI Quantum Team: Cirq. https://quantumai.google/cirq (2021), accessed: 13.08.2021
18. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper: a scalable quantum programming language. In: PLDI. pp. 333–342. ACM (2013)
19. Hietala, K., Rand, R., Hung, S., Li, L., Hicks, M.: Proving quantum programs correct. In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference). LIPIcs, vol. 193, pp. 21:1–21:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.21, https://doi.org/10.4230/LIPIcs.ITP.2021.21
20. Hietala, K., Rand, R., Hung, S., Wu, X., Hicks, M.: A verified optimizer for quantum circuits. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434318, https://doi.org/10.1145/3434318
21. 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, https://doi.org/10.1016/S0167-6423(99)00023-4
22. 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, https://doi.org/10.1145/3498687
23. Mac Lane, S.: Categories for the Working Mathematician (2nd ed.). Springer (1998)
24. McBride, C.: I got plenty o' nuttin'. In: Lindley, S., McBride, C., Trinder, P.W., Sannella, D. (eds.) A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 9600, pp. 207–233. Springer (2016). https://doi.org/10.1007/978-3-319-30936-1_12, https://doi.org/10.1007/978-3-319-30936-1_12
25. McClean, J.R., Romero, J., Babbush, R., Aspuru-Guzik, A.: The theory of variational hybrid quantum-classical algorithms. New Journal of Physics **18**(2), 023023 (2016)
26. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press (2010). https://doi.org/10.1017/CBO9780511976667

27. Paetznick, A., Svore, K.M.: Repeat-until-success: Non-deterministic decomposition of single-qubit unitaries. Quantum Info. Comput. **14**(15–16), 1277–1301 (Nov 2014)
28. Paykin, J., Rand, R., Zdancewic, S.: QWIRE: a core language for quantum circuits. In: POPL. pp. 846–858. ACM (2017)
29. Péchoux, R., Perdrix, S., Rennela, M., Zamdzhiev, V.: Quantum programming with inductive datatypes: Causality and affine type theory. In: Foundations of Software Science and Computation Structures, FOSSACS 2020. Lecture Notes in Computer Science, vol. 12077, pp. 562–581. Springer (2020). https://doi.org/10.1007/978-3-030-45231-5_29
30. Peruzzo, A., McClean, J., Shadbolt, P., Yung, M.H., Zhou, X.Q., Love, P.J., Aspuru-Guzik, A., O'brien, J.L.: A variational eigenvalue solver on a photonic quantum processor. Nature communications **5**(1), 1–7 (2014)
31. Rand, R., Paykin, J., Lee, D., Zdancewic, S.: ReQWIRE: Reasoning about reversible quantum circuits. In: Selinger, P., Chiribella, G. (eds.) Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Canada, 3-7th June 2018. EPTCS, vol. 287, pp. 299–312 (2018). https://doi.org/10.4204/EPTCS.287.17, https://doi.org/10.4204/EPTCS.287.17
32. Selinger, P., Valiron, B.: A lambda calculus for quantum computation with classical control. Mathematical Structures in Computer Science **16**(3), 527–552 (2006)
33. Seo, K.Y.: Indexed state monad blog post. https://kseo.github.io/posts/2017-01-12-indexed-monads.html (2017), accessed: 13.08.2021
34. 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
35. Svore, K., Geller, A., Troyer, M., Azariah, J., Granade, C., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A., Roetteler, M.: Q#: Enabling scalable quantum computing and development with a high-level dsl. In: Proceedings of the Real World Domain Specific Languages Workshop 2018. RWDSL2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3183895.3183901, https://doi.org/10.1145/3183895.3183901
36. Wootters, W.K., Zurek, W.H.: A single quantum cannot be cloned. Nature **299**(5886), 802–803 (1982)

Automatic Alignment in Higher-Order Probabilistic Programming Languages[⋆]

Daniel Lundén[1] (✉), Gizem Çaylak[1], Fredrik Ronquist[2,3], and David Broman[1]

[1] EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden, `{dlunde,caylak,dbro}@kth.se`
[2] Department of Bioinformatics and Genetics, Swedish Museum of Natural History, Stockholm, Sweden, `fredrik.ronquist@nrm.se`
[3] Department of Zoology, Stockholm University, Stockholm, Sweden

Abstract. Probabilistic Programming Languages (PPLs) allow users to encode statistical inference problems and automatically apply an *inference algorithm* to solve them. Popular inference algorithms for PPLs, such as sequential Monte Carlo (SMC) and Markov chain Monte Carlo (MCMC), are built around *checkpoints*—relevant events for the inference algorithm during the execution of a probabilistic program. Deciding the location of checkpoints is, in current PPLs, not done optimally. To solve this problem, we present a static analysis technique that automatically determines checkpoints in programs, relieving PPL users of this task. The analysis identifies a set of checkpoints that execute in the same order in every program run—they are *aligned*. We formalize alignment, prove the correctness of the analysis, and implement the analysis as part of the higher-order functional PPL Miking CorePPL. By utilizing the alignment analysis, we design two novel inference algorithm variants: *aligned SMC* and *aligned lightweight MCMC*. We show, through real-world experiments, that they significantly improve inference execution time and accuracy compared to standard PPL versions of SMC and MCMC.

Keywords: Probabilistic programming · Operational semantics · Static analysis.

1 Introduction

Probabilistic programming languages (PPLs) are languages used to encode statistical inference problems, common in research fields such as phylogenetics [39],

[⋆] This project is financially supported by the Swedish Foundation for Strategic Research (FFL15-0032 and RIT15-0012), and also partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and the Swedish Research Council (grants 2018-04620 and 2021-04830). 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.

The original version of this chapter was revised: Theorem 1 has been corrected. The correction to this chapter is available at https://doi.org/10.1007/978-3-031-30044-8_21

computer vision [16], topic modeling [5], data cleaning [23], and cognitive science [15]. PPL implementations automatically solve encoded problems by applying an *inference algorithm*. In particular, automatic inference allows users to solve inference problems without having in-depth knowledge of inference algorithms and how to apply them. Some examples of PPLs are WebPPL [14], Birch [31], Anglican [48], Miking CorePPL [25], Turing [12], and Pyro [3].

Sequential Monte Carlo (SMC) and Markov chain Monte Carlo (MCMC) are general-purpose families of inference algorithms often used for PPL implementations. These algorithms share the concept of *checkpoints*: relevant execution events for the inference algorithm. For SMC, the checkpoints are *likelihood updates* [48,14] and determine the *resampling* of executions. Alternatively, users must sometimes manually annotate or write the probabilistic program in a certain way to make resampling explicit [25,31]. For MCMC, checkpoints are instead *random draws*, which allow the inference algorithm to manipulate these draws to construct a Markov chain over program executions [47,38]. When designing SMC and MCMC algorithms for *universal PPLs*[4], both the *placement* and *handling* of checkpoints are critical to making the inference both efficient and accurate.

For SMC, a standard inference approach is to resample at *all* likelihood updates [14,48]. This approach produces correct results asymptotically [24] but is highly problematic for certain models [39]. Such models require non-trivial and SMC-specific manual program rewrites to force good resampling locations and make SMC tractable. Overall, choosing the likelihood updates at which to resample significantly affects SMC execution time and accuracy.

For MCMC, a standard approach for inference in universal PPLs is *lightweight MCMC* [47], which constructs a Markov chain over random draws in programs. The key idea is to use an *addressing transformation* and a *runtime database* of random draws. Specifically, the database enables matching and reusing random draws between executions according to their *stack traces*, even if the random draws may or may not occur due to randomness during execution. However, the dynamic approach of looking up random draws in the database through their stack traces is expensive and introduces significant runtime overhead.

To overcome the SMC and MCMC problems in universal PPLs, we present a static analysis technique for higher-order functional PPLs that *automatically* determines checkpoints in a probabilistic program that always occur in the same order in every program execution—they are *aligned*. We formally define alignment, formalize the alignment analysis, and prove the soundness of the analysis with respect to the alignment definition. The novelty and challenge in developing the static analysis technique is to capture alignment properties through the identification of expressions in programs that may evaluate to *stochastic values* and expressions that may evaluate due to *stochastic branching*. Stochastic branching results from `if` expressions with stochastic values as conditions or function applications where the function itself is stochastic. Stochastic values and branches pose a significant challenge when proving the soundness of the analysis.

[4] A term coined by Goodman et al. [13]. Essentially, it means that the types and numbers of random variables cannot be determined statically.

We design two new inference algorithms that improve accuracy and execution time compared to current approaches. Unlike the standard SMC algorithm for PPLs [48,14], *aligned SMC* only resamples at aligned likelihood updates. Resampling only at aligned likelihood updates guarantees that each SMC execution resamples the same number of times, which makes expensive global termination checks redundant [25]. We evaluate aligned SMC on two diversification models from Ronquist et al. [39] and a state-space model for aircraft localization, demonstrating significantly improved inference accuracy and execution time compared to traditional SMC. Both models—constant rate birth-death (CRBD) and cladogenetic diversification rate shift (ClaDS)—are used in real-world settings and are of considerable interest to evolutionary biologists [33,28]. The documentations of both Anglican [48] and Turing [12] acknowledge the importance of alignment for SMC and state that all likelihood updates must be aligned. However, Turing and Anglican neither formalize nor enforce this property—it is up to the users to *manually* guarantee it, often requiring non-standard program rewrites [39].

We also design *aligned lightweight MCMC*, a new version of lightweight MCMC [47]. Aligned lightweight MCMC constructs a Markov chain over the program using the aligned random draws as *synchronization points* to match and reuse aligned random draws and a subset of unaligned draws between executions. Aligned lightweight MCMC does not require a runtime database of random draws and therefore reduces runtime overhead. We evaluate aligned lightweight MCMC for latent Dirichlet allocation (LDA) [5] and CRBD [39], demonstrating significantly reduced execution times and no decrease in inference accuracy. Furthermore, automatic alignment is orthogonal to and easily combines with the lightweight MCMC optimizations introduced by Ritchie et al. [38].

We implement the analysis, aligned SMC, and aligned lightweight MCMC in Miking CorePPL [25,7]. In addition to analyzing stochastic `if`-branching, the implementation analyzes stochastic branching at a standard pattern-matching construct. Compared to `if` expressions, the pattern-matching construct requires a more sophisticated analysis of the pattern and the value matched against it to determine if the pattern-matching causes a stochastic branch.

In summary, we make the following contributions.

- We invent and formalize alignment for PPLs. Aligned parts of a program occur in the same order in every execution (Section 4.1).
- We formalize and prove the soundness of a novel static analysis technique that determines stochastic value flow and stochastic branching, and in turn alignment, in higher-order probabilistic programs (Section 4.2).
- We design aligned SMC inference that only resamples at aligned likelihood updates, improving execution time and inference accuracy (Section 5.1).
- We design aligned lightweight MCMC inference that only reuses aligned random draws, improving execution time (Section 5.2).
- We implement the analysis and inference algorithms in Miking CorePPL. The implementation extends the alignment analysis to identify stochastic branching resulting from pattern matching (Section 6).

Section 7 describes the evaluation and discusses its results. The paper also has an accompanying artifact that supports the evaluation [26]. Section 8 discusses related work and Section 9 concludes. Next, Section 2 considers a simple motivating example to illustrate the key ideas. Section 3 introduces syntax and semantics for the calculus used to formalize the alignment analysis.

An extended version of the paper is also available at arXiv [27]. 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 presents a motivating example that illustrates the key alignment ideas in relation to aligned SMC (Section 2.1) and aligned lightweight MCMC (Section 2.2). We assume basic knowledge of probability theory. Knowledge of PPLs is helpful, but not a strict requirement. The book by van de Meent et al. [46] provides a good introduction to PPLs.

Probabilistic programs encode Bayesian statistical inference problems with two fundamental constructs: `assume` and `weight`. The `assume` construct defines random variables, which make execution nondeterministic. Intuitively, a probabilistic program then encodes a probability distribution over program executions (the prior distribution), and it is possible to sample from this distribution by executing the program with random sampling at `assume`s. The `weight` construct updates the *likelihood* of individual executions. Updating likelihoods for executions modifies the probability distribution induced by `assume`s, and the inference problem encoded by the program is to determine or approximate this modified distribution (the posterior distribution). The main purpose of `weight` in real-world models is to condition executions on observed data.[5]

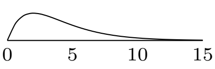
Consider the probabilistic program in Fig. 1a. The program is contrived and purposefully constructed to compactly illustrate alignment, but the real-world diversification models in Ronquist et al. [39] that we also consider in Section 7 inspired the program's general structure. The program defines (line 1) and returns (line 18) a Gamma-distributed random variable *rate*. Figure 1b illustrates the Gamma distribution. To modify the likelihood for values of *rate*, the program executes the *iter* function (line 10) three times, and the *survives* function (line 2) a random number of times *n* (line 13) within each *iter* call.

Conceptually, to infer the posterior distribution of the program, we execute the program infinitely many times. In each execution, we draw samples for the random variables defined at `assume`, and accumulate the likelihood at `weight`. The return value of the execution, weighted by the accumulated likelihood, represents one sample from the posterior distribution. Fig. 1c shows a histogram of such weighted samples of *rate* resulting from a large number of executions of Fig. 1a. The fundamental inference algorithm that produces such weighted samples is called *likelihood weighting* (a type of *importance sampling* [32]). We

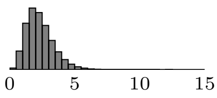
[5] A number of more specialized constructs for likelihood updating are also available in various PPLs, for example *observe* [48,14] and *condition* [14].

```
 1 let rate = assume Gamma(2, 2) in
 2 let rec survives = λn.
 3   if n = 0 then () else
 4     if assume Bernoulli(0.9) then
 5       weight 0.5;
 6       survives (n − 1)
 7     else
 8       weight 0
 9 in
10 let rec iter = λi.
11   if i = 0 then () else
12     weight rate;
13     let n = assume Poisson(rate) in
14     survives n;
15     iter (i − 1)
16 in
17 iter 3;
18 rate
```

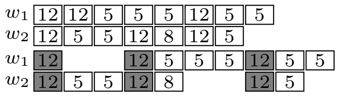
(a) Probabilistic program.



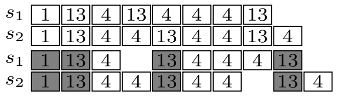
(b) Gamma(2, 2).



(c) Histogram.



(d) Aligning `weight`.



(e) Aligning `assume`.

Fig. 1: A simple example illustrating alignment. Fig. (a) gives a probabilistic program using functional-style PPL pseudocode. Fig. (b) illustrates the Gamma(2, 2) probability density function. Fig. (c) illustrates a histogram over weighted *rate* samples produced by running the program in (a) a large number of times. Fig. (d) shows two line number sequences w_1 and w_2 of `weight`s encountered in two program runs (top) and how to align them (bottom). Fig. (e) shows two line number sequences s_1 and s_2 of `assume`s encountered in two program runs (top) and how to align them (bottom).

see that, compared to the prior distribution for *rate* in Fig. 1b, the posterior is more sharply peaked due to the likelihood modifications.

2.1 Aligned SMC

Likelihood weighting can only handle the simplest of programs. In Fig. 1a, a problem with likelihood weighting is that we assign the weight 0 to many executions at line 8. These executions contribute nothing to the final distribution. SMC solves this by executing many program instances *concurrently* and occasionally *resampling* them (with replacement) based on their current likelihoods. Resampling discards executions with lower weights (in the worst case, 0) and replaces them with executions with higher weights. The most common approach in popular PPLs is to resample just after likelihood updates (i.e., calls to `weight`).

Resampling at all calls to `weight` in Fig. 1a is suboptimal. The best option is instead to *only* resample at line 12. This is because executions encounter lines 5 and 8 a *random* number of times due to the stochastic branch at line 3, while they encounter line 12 a fixed number of times. As a result of resampling at lines 5 and 8, executions become *unaligned*; in each resampling, executions can have reached either line 5, line 8, or line 12. On the other hand, if we resample only at line 12, all executions will always have reached line 12 for the same iteration of *iter* in every resampling. Intuitively, this is a sensible approach since, when resampling, executions have progressed the same distance through the program. We say that the `weight` at line 12 is *aligned*, and resampling only at aligned `weight`s results in our new inference approach called *aligned SMC*. Fig. 1d visualizes the `weight` alignment for two sample executions of Fig. 1a.

2.2 Aligned Lightweight MCMC

Another improvement over likelihood weighting is to construct a Markov chain over program executions. It is beneficial to propose new executions in the Markov chain by making small, rather than large, modifications to the previous execution. The lightweight MCMC [47] algorithm does this by redrawing a single random draw in the previous execution, and then reusing as many other random draws as possible. Random draws in the current and previous executions match through *stack traces*—the sequences of applications leading up to a random draw. Consider the random draw at line 13 in Fig. 1a. It is called exactly three times in every execution. If we identify applications and `assume`s by line numbers, we get the stack traces $[17, 13]$, $[17, 15, 13]$, and $[17, 15, 15, 13]$ for these three `assume`s in every execution. Consequently, lightweight MCMC can reuse these draws by storing them in a database indexed by stack traces.

The stack trace indexing in lightweight MCMC is overly complicated when reusing aligned random draws. Note that the `assume`s at lines 1 and 13 in Fig 1a are aligned, while the `assume` at line 4 is unaligned. Fig. 1e visualizes the `assume` alignment for two sample executions of Fig. 1a. Aligned random draws occur in the same same order in every execution, and are therefore trivial to match and reuse between executions through indexing by counting. The appeal with stack trace indexing is to additionally allow reusing a subset of *unaligned* draws.

A key insight in this paper is that aligned random draws can also act as *synchronization points* in the program to allow reusing unaligned draws *without* a stack trace database. After an aligned draw, we reuse unaligned draws occurring up until the next aligned draw, as long as they syntactically originate at the same `assume` as the corresponding unaligned draws in the previous execution. As soon as an unaligned draw does not originate from the same `assume` as in the previous execution, we redraw all remaining unaligned draws up until the next aligned draw. Instead of a trace-indexed database, this approach requires storing a list of unaligned draws (tagged with identifiers of the `assume`s at which they originated) for each execution segment in between aligned random draws. For example, for the execution s_1 in Fig. 1e, we store lists of unaligned Bernoulli random draws from line 4 for each execution segment in between the three aligned random draws at line 13. If a Poisson random draw n at line 13 does not change or decreases, we can reuse the stored unaligned Bernoulli draws up until the next Poisson random draw as *survives* executes n or fewer times. If the drawn n instead increases to n', we can again reuse all stored Bernoulli draws, but must supplement them with new Bernoulli draws to reach n' draws in total.

As we show in Section 7, using aligned draws as synchronization points works very well in practice and avoids the runtime overhead of the lightweight MCMC database. However, manually identifying aligned parts of programs and rewriting them so that inference can make use of alignment is, if even possible, tedious, error-prone, and impractical for large programs. This paper presents an automated approach to identifying aligned parts of programs. Combining static alignment analysis and using aligned random draws as synchronization points form the key ideas of the new algorithm that we call *aligned lightweight MCMC*.

3 Syntax and Semantics

In preparation for the alignment analysis in Section 4, we require an idealized base calculus capturing the key features of expressive PPLs. This section introduces such a calculus with a formal syntax (Section 3.1) and semantics (Section 3.2). We assume a basic understanding of the lambda calculus (see, e.g., Pierce [37] for a complete introduction). Section 6 further describes extending the idealized calculus and the analysis in Section 4 to a full-featured PPL.

3.1 Syntax

We use the untyped lambda calculus as the base for our calculus. We also add `let` expressions for convenience, and `if` expressions to allow intrinsic booleans to affect control flow. The calculus is a subset of the language used in Fig. 1a. We inductively define terms $\mathbf{t}$ and values $\mathbf{v}$ as follows.

Definition 1 (Terms and values).

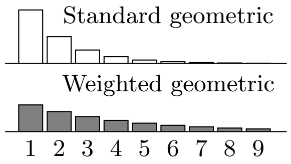
$$\begin{aligned}\mathbf{t} ::= {}& x \mid c \mid \lambda x.\ \mathbf{t} \mid \mathbf{t}\ \mathbf{t} \mid \texttt{let}\ x = \mathbf{t}\ \texttt{in}\ \mathbf{t} \qquad\qquad \mathbf{v} ::= c \mid \langle \lambda x.\ \mathbf{t}, \rho \rangle \\ & \mid \texttt{if}\ \mathbf{t}\ \texttt{then}\ \mathbf{t}\ \texttt{else}\ \mathbf{t} \mid \texttt{assume}\ \mathbf{t} \mid \texttt{weight}\ \mathbf{t} \\ & x, y \in X \quad \rho \in P \quad c \in C \quad \{\text{false}, \text{true}, ()\} \cup \mathbb{R} \cup D \subseteq C.\end{aligned} \tag{1}$$

X is a countable set of variable names, C a set of intrinsic values and operations, and $D \subset C$ a set of probability distributions. The set P contains all evaluation environments ρ, that is, partial functions mapping names in X to values $\mathbf{v}$. We use T and V to denote the set of all terms and values, respectively.

Values $\mathbf{v}$ are intrinsics or closures, where closures are abstractions with an environment binding free variables in the abstraction body. We require that C include booleans, the unit value $()$, and real numbers. The reason is that `weight` takes real numbers as argument and returns $()$ and that `if` expression conditions are booleans. Furthermore, probability distributions are often over booleans and real numbers. For example, we can include the normal distribution constructor $\mathcal{N} \in C$ that takes real numbers as arguments and produces normal distributions over real numbers. For example, $\mathcal{N}\ 0\ 1 \in D$, the standard normal distribution. We often write functions in C in infix position or with standard function application syntax for readability. For example, $1 + 2$ with $+ \in C$ means $+\ 1\ 2$, and $\mathcal{N}(0, 1)$ means $\mathcal{N}\ 0\ 1$. Additionally, we use the shorthand $\mathbf{t}_1; \mathbf{t}_2$ for `let _ =` $\mathbf{t}_1$ `in` $\mathbf{t}_2$, where _ is the do-not-care symbol. That is, $\mathbf{t}_1; \mathbf{t}_2$ evaluates $\mathbf{t}_1$ for side

```
1 let rec geometric = λ_.
2   let x = assume Bernoulli(0.5) in
3   if x then
4     weight 1.5;
5     1 + geometric ()
6   else 1
7 in geometric ()
```

(a) Probabilistic program $\mathbf{t}_{geo}$.



(b) Probability distributions.

Fig. 2: A probabilistic program $\mathbf{t}_{geo}$ [25], illustrating (1). Fig. (a) gives the program, and (b) the corresponding probability distributions. In (b), the y-axis gives the probability, and the x-axis gives the outcome (the number of coin flips). The upper part of (b) excludes the shaded `weight` at line 4 in (a).

effects only before evaluating $\mathbf{t}_2$. Finally, the untyped lambda calculus supports recursion through fixed-point combinators. We encapsulate this in the shorthand `let rec` f `=` $\lambda x.\mathbf{t}_1$ `in` $\mathbf{t}_2$ to conveniently define recursive functions.

The `assume` and `weight` constructs are PPL-specific. We define random variables from intrinsic probability distributions with `assume` (also known as *sample* in PPLs with sampling-based inference). For example, the term `let` x `=` `assume` $\mathcal{N}(0,1)$ `in` $\mathbf{t}$ defines x as a random variable with a standard normal distribution in $\mathbf{t}$. Boolean random variables combined with `if` expressions result in *stochastic branching*—causing the alignment problem. Lastly, `weight` (also known as *factor* or *score*) is a standard construct for likelihood updating (see, e.g., Borgström et al. [6]). Next, we illustrate and formalize a semantics for (1).

3.2 Semantics

Consider the small probabilistic program $\mathbf{t}_{geo} \in T$ in Fig. 2a. The program encodes the standard geometric distribution via a function *geometric*, which recursively flips a fair coin (a Bernoulli(0.5) distribution) at line 2 until the outcome is false (i.e., tails). At that point, the program returns the total number of coin flips, including the last tails flip. The upper part of Fig. 2b illustrates the result distribution for an infinite number of program runs with line 4 ignored.

To illustrate the effect of `weight`, consider $\mathbf{t}_{geo}$ with line 4 included. This `weight` modifies the likelihood with a factor 1.5 each time the flip outcome is true (or, heads). Intuitively, this emphasizes larger return values, illustrated in the lower part of Fig. 2b. Specifically, the (unnormalized) probability of seeing n coin flips is $0.5^n \cdot 1.5^{n-1}$, compared to 0.5^n for the unweighted version. The factor 1.5^{n-1} is the result of the calls to `weight`.

We now introduce a big-step operational semantics for single runs of programs $\mathbf{t}$. Such a semantics is essential to formalize the probability distributions encoded by probabilistic programs (e.g., Fig. 2b for Fig. 2a) and to prove the correctness of PPL inference algorithms. For example, Borgström et al. [6] define a PPL calculus and semantics similar to this paper and formally proves the correctness of an MCMC algorithm. Another example is Lundén et al. [24], who also define a

$$\frac{}{\rho \vdash x \;{}^{[]}\!\Downarrow^{1}_{[]}\; \rho(x)}\,(\textsc{Var}) \qquad \frac{}{\rho \vdash c \;{}^{[]}\!\Downarrow^{1}_{[]}\; c}\,(\textsc{Const}) \qquad \frac{}{\rho \vdash \lambda x.\mathbf{t} \;{}^{[]}\!\Downarrow^{1}_{[]}\; \langle \lambda x.\mathbf{t}, \rho \rangle}\,(\textsc{Lam})$$

$$\frac{\rho \vdash \mathbf{t}_1 \;{}^{s_1}\!\Downarrow^{w_1}_{l_1}\; \langle \lambda x.\mathbf{t}, \rho' \rangle \quad \rho \vdash \mathbf{t}_2 \;{}^{s_2}\!\Downarrow^{w_2}_{l_2}\; \mathbf{v}_2 \quad \rho', x \mapsto \mathbf{v}_2 \vdash \mathbf{t} \;{}^{s_3}\!\Downarrow^{w_3}_{l_3}\; \mathbf{v}}{\rho \vdash \mathbf{t}_1 \;\; \mathbf{t}_2 \;{}^{s_1 \| s_2 \| s_3}\!\Downarrow^{w_1 \cdot w_2 \cdot w_3}_{l_1 \| l_2 \| l_3}\; \mathbf{v}}\,(\textsc{App})$$

$$\frac{\rho \vdash \mathbf{t}_1 \;{}^{s_1}\!\Downarrow^{w_1}_{l_1}\; c_1 \quad \rho \vdash \mathbf{t}_2 \;{}^{s_2}\!\Downarrow^{w_2}_{l_2}\; c_2}{\rho \vdash \mathbf{t}_1 \;\; \mathbf{t}_2 \;{}^{s_1 \| s_2}\!\Downarrow^{w_1 \cdot w_2}_{l_1 \| l_2}\; \delta(c_1, c_2)}\,(\textsc{Const-App}) \qquad \frac{\rho \vdash \mathbf{t} \;{}^{s}\!\Downarrow^{w}_{l}\; d \quad w' = f_d(c)}{\rho \vdash \texttt{assume}\;\mathbf{t} \;{}^{s \| [c]}\!\Downarrow^{w \cdot w'}_{l}\; c}\,(\textsc{Assume})$$

$$\frac{\rho \vdash \mathbf{t}_1 \;{}^{s_1}\!\Downarrow^{w_1}_{l_1}\; \mathbf{v}_1 \quad \rho, x \mapsto \mathbf{v}_1 \vdash \mathbf{t}_2 \;{}^{s_2}\!\Downarrow^{w_2}_{l_2}\; \mathbf{v}}{\rho \vdash \texttt{let}\; x = \mathbf{t}_1 \;\texttt{in}\; \mathbf{t}_2 \;{}^{s_1 \| s_2}\!\Downarrow^{w_1 \cdot w_2}_{l_1 \| [x] \| l_2}\; \mathbf{v}}\,(\textsc{Let}) \qquad \frac{\rho \vdash \mathbf{t} \;{}^{s}\!\Downarrow^{w}_{l}\; w'}{\rho \vdash \texttt{weight}\;\mathbf{t} \;{}^{s}\!\Downarrow^{w \cdot w'}_{l}\; ()}\,(\textsc{Weight})$$

$$\frac{\rho \vdash \mathbf{t}_1 \;{}^{s_1}\!\Downarrow^{w_1}_{l_1}\; \text{true} \quad \rho \vdash \mathbf{t}_2 \;{}^{s_2}\!\Downarrow^{w_2}_{l_2}\; \mathbf{v}_2}{\rho \vdash \texttt{if}\; \mathbf{t}_1 \;\texttt{then}\; \mathbf{t}_2 \;\texttt{else}\; \mathbf{t}_3 \;{}^{s_1 \| s_2}\!\Downarrow^{w_1 \cdot w_2}_{l_1 \| l_2}\; \mathbf{v}_2}\,(\textsc{If-True})$$

$$\frac{\rho \vdash \mathbf{t}_1 \;{}^{s_1}\!\Downarrow^{w_1}_{l_1}\; \text{false} \quad \rho \vdash \mathbf{t}_3 \;{}^{s_3}\!\Downarrow^{w_3}_{l_3}\; \mathbf{v}_3}{\rho \vdash \texttt{if}\; \mathbf{t}_1 \;\texttt{then}\; \mathbf{t}_2 \;\texttt{else}\; \mathbf{t}_3 \;{}^{s_1 \| s_3}\!\Downarrow^{w_1 \cdot w_3}_{l_1 \| l_3}\; \mathbf{v}_3}\,(\textsc{If-False})$$

Fig. 3: A big-step operational semantics for terms, formalizing single runs of programs $\mathbf{t} \in T$. The operation $\rho, x \mapsto \mathbf{v}$ produces a new environment extending ρ with a binding $\mathbf{v}$ for x. For each distribution $d \in D$, f_d is its *probability density* or *probability mass* function—encoding the relative probability of drawing particular values from the distribution. For example, $f_{\text{Bernoulli}(0.3)}(\text{true}) = 0.3$ and $f_{\text{Bernoulli}(0.3)}(\text{false}) = 1 - 0.3 = 0.7$. We use $\cdot$ to denote multiplication.

similar calculus and semantics and prove the correctness of PPL SMC algorithms. In particular, the correctness of our aligned SMC algorithm (Section 5.1) follows from this proof. The purpose of the semantics in this paper is to formalize alignment and prove the soundness of our analysis in Section 4. We use a big-step semantics as the finer granularity in a small-step semantics is redundant. We begin with a definition for intrinsics.

Definition 2 (Intrinsic functions). *For every $c \in C$, we attach an* arity *$|c| \in \mathbb{N}$. We define a partial function $\delta : C \times C \to C$ such that $\delta(c, c_1) = c_2$ is defined for $|c| > 0$. For all c, c_1, and c_2, such that $\delta(c, c_1) = c_2$, $|c_2| = |c| - 1$.*

Intrinsic functions are curried and produce intrinsic or intrinsic functions of one arity less through δ. For example, for $+ \in C$, we have $\delta(\delta(+, 1), 2) = 3$, $|+| = 2$, $|\delta(+, 1)| = 1$, and $|\delta(\delta(+, 1), 2)| = 0$. Next, randomness in our semantics is deterministic via a *trace* of random draws in the style of Kozen [22].

Definition 3 (Traces). *The set S of traces is the set such that, for all $s \in S$, s is a sequence of intrinsics from C with arity 0.*

In the following, we use the notation $[c_1, c_2, \ldots, c_n]$ for sequences and $\|$ for sequence concatenation. For example, $[c_1, c_2] \| [c_2, c_4] = [c_1, c_2, c_3, c_4]$. We also use subscripts to select elements in a sequence, e.g., $[c_1, c_2, c_3, c_4]_2 = c_2$. In practice, traces are often sequences of real numbers, e.g., $[1.1, 3.2, 8.4] \in S$.

Fig. 3 presents the semantics as a relation $\rho \vdash \mathbf{t} \; {}^{s}\!\Downarrow_{l}^{w} \; \mathbf{v}$ over $P \times T \times S \times \mathbb{R} \times L \times V$. L is the set of sequences over X, i.e., sequences of names. For example, $[x, y, z] \in L$, where $x, y, z \in X$. We use $l \in L$ to track the sequence of `let`-bindings during evaluation. For example, evaluating `let` x `= 1 in let` y `= 2 in` $x + y$ results in $l = [x, y]$. In Section 4, we use the sequence of encountered `let`-bindings to define alignment. For simplicity, from now on we assume that bound variables are always unique (i.e., variable shadowing is impossible).

It is helpful to think of ρ, $\mathbf{t}$, and s as the input to $\Downarrow$, and l, w and $\mathbf{v}$ as the output. In the environment ρ, $\mathbf{t}$, with trace s, evaluates to $\mathbf{v}$, encounters the sequence of `let` bindings l, and accumulates the weight w. The trace s is the sequence of all random draws, and each random draw in (Assume) consumes precisely one element of s. The rule (Let) tracks the sequence of bindings by adding x at the correct position in l. The number w is the likelihood of the execution—the probability density of all draws in the program, registered at (Assume), combined with direct likelihood modifications, registered at (Weight). The remaining aspects of the semantics are standard (see, e.g., Kahn [20]). To give an example of the semantics, we have $\varnothing \vdash \mathbf{t}_{geo} \; {}^{[\text{true},\text{true},\text{true},\text{false}]}\!\Downarrow_{[geometric,x,x,x,x]}^{0.5\cdot 1.5\cdot 0.5\cdot 1.5\cdot 0.5\cdot 1.5\cdot 0.5} \; 4$ for the particular execution of $\mathbf{t}_{geo}$ making three recursive calls. Next, we formalize and apply the alignment analysis to (1).

4 Alignment Analysis

This section presents the main contribution of this paper: automatic alignment in PPLs. Section 4.1 introduces A-normal form and gives a precise definition of alignment. Section 4.2 formalizes and proves the correctness of the alignment analysis. Lastly, Section 4.3 discusses a dynamic version of alignment.

4.1 A-Normal Form and Alignment

To reason about all subterms $\mathbf{t}'$ of a program $\mathbf{t}$ and to enable the analysis in Section 4.2, we need to uniquely label all subterms. A straightforward approach is to use variable names within the program itself as labels (remember that we assume bound variables are always unique). This leads us to the standard A-normal form (ANF) representation of programs [11].

Definition 4 (A-normal form).

$$\begin{aligned} \mathbf{t}_{\text{ANF}} &::= x \mid \texttt{let } x = \mathbf{t}'_{\text{ANF}} \texttt{ in } \mathbf{t}_{\text{ANF}} \\ \mathbf{t}'_{\text{ANF}} &::= x \mid c \mid \lambda x.\ \mathbf{t}_{\text{ANF}} \mid x\ y \\ &\quad \mid \texttt{if } x \texttt{ then } \mathbf{t}_{\text{ANF}} \texttt{ else } \mathbf{t}_{\text{ANF}} \mid \texttt{assume } x \mid \texttt{weight } x \end{aligned} \tag{2}$$

We use T_{ANF} to denote the set of all terms $\mathbf{t}_{\text{ANF}}$. Unlike $\mathbf{t} \in T$, $\mathbf{t}_{\text{ANF}} \in T_{\text{ANF}}$ enforces that a variable bound by a `let` labels each subterm in the program. Furthermore, we can automatically transform any program in T to a semantically equivalent T_{ANF} program, and $T_{\text{ANF}} \subset T$. Therefore, we assume in the remainder of the paper that all terms are in ANF.

Given the importance of alignment in universal PPLs, it is somewhat surprising that there are no previous attempts to give a formal definition of its meaning. Here, we give a first such formal definition, but before defining alignment, we require a way to restrict, or filter, sequences.

Definition 5 (Restriction of sequences). *For all $l \in L$ and $Y \subseteq X$, $l|_Y$ (the restriction of l to Y) is the subsequence of l with all elements not in Y removed.*

For example, $[x, y, z, y, x]|_{\{x,z\}} = [x, z, x]$. We now formally define alignment.

Definition 6 (Alignment). *For $\mathbf{t} \in T_{\text{ANF}}$, let $X_{\mathbf{t}}$ denote all variables that occur in $\mathbf{t}$. The sets $A_{\mathbf{t}} \in \mathcal{A}_{\mathbf{t}}$, $A_{\mathbf{t}} \subseteq X_{\mathbf{t}}$, are the largest sets such that, for arbitrary $\varnothing \vdash \mathbf{t} \; {}^{s_1}\!\Downarrow^{w_1}_{l_1} \mathbf{v}_1$ and $\varnothing \vdash \mathbf{t} \; {}^{s_2}\!\Downarrow^{w_2}_{l_2} \mathbf{v}_2$, $l_1|_{A_{\mathbf{t}}} = l_2|_{A_{\mathbf{t}}}$.*

For a given $A_{\mathbf{t}}$, the *aligned expressions*—expressions bound by a `let` to a variable name in $A_{\mathbf{t}}$—are those that occur in the same order in every execution, regardless of random draws. We seek the largest sets, as $A_{\mathbf{t}} = \varnothing$ is always a trivial solution. Assume we have a program with $X_{\mathbf{t}} = \{x, y, z\}$ and such that $l = [x, y, x, z, x]$ and $l = [x, y, x, z, x, y]$ are the only possible sequences of `let` bindings. Then, $A_{\mathbf{t}} = \{x, z\}$ is the only possibility. It is also possible to have multiple choices for $A_{\mathbf{t}}$. For example, if $l = [x, y, z]$ and $l = [x, z, y]$ are the only possibilities, then $\mathcal{A}_{\mathbf{t}} = \{\{x, z\}, \{x, y\}\}$. Next, assume that we transform the programs in Fig. 2a and Fig. 1a to ANF. The expression labeled by x in Fig. 2a is then clearly not aligned, as random draws determine how many times it executes (l could be, e.g., $[x, x]$ or $[x, x, x, x]$). Conversely, the expression n (line 13) in Fig. 1a is aligned, as its number and order of evaluations do not depend on any random draws.

Definition 6 is *context insensitive*: for a given $A_{\mathbf{t}}$, each x is either aligned or unaligned. One could also consider a context-sensitive definition of alignment in which x can be aligned in some contexts and unaligned in others. A context could, for example, be the sequence of function applications (i.e., the call stack) leading up to an expression. Considering different contexts for x is complicated and difficult to take full advantage of. We justify the choice of context-insensitive alignment with the real-world models in Section 7, neither of which requires a context-sensitive alignment.

With alignment defined, we now move on to the static alignment analysis.

4.2 Alignment Analysis

The basis for the alignment analysis is 0-CFA [34,42]—a static analysis framework for higher-order functional programs. The prefix 0 indicates that 0-CFA is context insensitive. There is also a set of analyses k-CFA [30] that adds increasing amounts (with $k \in \mathbb{N}$) of context sensitivity to 0-CFA. We could use such analyses with a context-sensitive version of Definition 6. However, the potential benefit of k-CFA is also offset by the worst-case exponential time complexity, already at $k = 1$. In contrast, the time complexity of 0-CFA is polynomial (cubic in the worst-case). The alignment analysis for the models in Section 7 runs instantaneously, justifying that the time complexity is not a problem in practice.

```
 1 let n1 = ¬ in let n2 = ¬ in
 2 let one = 1 in
 3 let half = 0.5 in let c = true in
 4 let f1 = λx1. let t1 = weight one in x1 in
 5 let f2 = λx2. let t2 = weight one in t2 in
 6 let f3 = λx3. let t3 = weight one in t3 in
 7 let f4 = λx4. let t4 = weight one in t4 in
 8 let bern = Bernoulli in
 9 let d1 = bern half in
10 let a1 = assume d1
11 let v1 = f1 one in
12 let v2 = n1 a1 in
13 let v3 = n2 c in
14 let f5 =
15   if a1 then let t5 = f4 one in f2
16   else f3
17 in
18 let v4 = f5 one in
19 let i1 =
20   if c then let t6 = f1 one in t6
21   else one
22 in i1
```

Fig. 4: A program $\mathbf{t}_{example} \in T_{\text{ANF}}$ illustrating the analysis.

The extensions to 0-CFA required to analyze alignment are non-trivial to design, but the resulting formalization is surprisingly simple. The challenge is instead to prove that the extensions correctly capture the alignment property from Definition 6. We extend 0-CFA to analyze stochastic values and alignment in programs $\mathbf{t} \in T_{\text{ANF}}$. As with most static analyses, our analysis is sound but conservative (i.e., sound but incomplete)—the analysis may mark aligned expressions of programs as unaligned, but not vice versa. That the analysis is conservative does not degrade the alignment analysis results for any model in Section 7, which justifies the approach. We divide the formal analysis into two algorithms. Algorithm 1 generates *constraints* for $\mathbf{t}$ that a valid analysis solution must satisfy. This section describes Algorithm 1 and the generated constraints. The second algorithm computes a solution that satisfies the generated constraints. We describe the algorithm at a high level, but omit a full formalization.[†]

For soundness of the analysis, we require $\langle \lambda x.\ \mathbf{t}, \rho \rangle \notin C$ (recall that C is the set of intrinsics). That is, closures are *not* in C. By Definition 3, this implies that closures are not in the sample space of probability distributions in D and that evaluating intrinsics never produces closures (this would unnecessarily complicate the analysis without any benefit).

In addition to standard 0-CFA constraints, Algorithm 1 generates new constraints for *stochastic values* and *unalignment*. We use the contrived but illustrative program in Fig. 4 as an example. Note that, while omitted from Fig. 4 for ease of presentation, the analysis also supports recursion introduced through `let rec`. Stochastic values are values in the program affected by random variables. Stochastic values initially originate at `assume` and then propagate through programs via function applications and `if` expressions. For example, a_1 (line 10) is stochastic because of `assume`. We subsequently use a_1 to define v_2 via n_1 (line 12), which is then also stochastic. Similarly, a_1 is the condition for the `if` resulting in f_5 (line 14), and the function f_5 is therefore also stochastic. When we apply f_5, it results in yet another stochastic value, v_4 (line 18). In conclusion, the stochastic values are a_1, v_2, f_5, and v_4.

Consider the flow of unalignment in Fig. 4. We mark expressions that may execute due to stochastic branching as unaligned. From our analysis of stochastic values, the program's only stochastic `if` condition is at line 15, and we determine that all expressions directly within the branches are unaligned. That is, the expression labeled by t_5 is unaligned. Furthermore, we apply the variable f_4 when defining t_5. Thus, *all* expressions in bodies of lambdas that flow to f_4 are unaligned. Here, it implies that t_4 is unaligned. Finally, we established that the function f_5 produced at line 15 is stochastic. Due to the application at line 18, all names bound by `lets` in bodies of lambdas that flow to f_5 are unaligned. Here, it implies that t_2 and t_3 are unaligned. In conclusion, the unaligned expressions are named by t_2, t_3, t_4, and t_5. For example, aligned SMC therefore resamples at the `weight` at t_1, but not at the `weights` at t_2, t_3, and t_4.

Consider the program in Fig. 1a again, and assume it is transformed to ANF. The alignment analysis must mark all names bound within the stochastic `if` at line 3 as unaligned because a stochastic value flows to its condition. In particular, the `weight` expressions at lines 5 and 8 are unaligned (and the `weight` at line 12 is aligned). Thus, aligned SMC resamples only at line 12.

To formalize the flow of stochastic values, we define *abstract values* $\mathbf{a} ::= \lambda x.y \mid$ `stoch` $\mid$ `const` n, where $x, y \in X$ and $n \in \mathbb{N}$. We use A to denote the set of all abstract values. The `stoch` abstract value is new and represents stochastic values. The $\lambda x.y$ and `const` n abstract values are standard and represent abstract closures and intrinsics, respectively. For each variable name x in the program, we define a set S_x containing abstract values that may occur at x. For example, in Fig. 4, we have `stoch` $\in S_{a_1}$, $(\lambda x_2.t_2) \in S_{f_2}$, and (`const` $1) \in S_{n_1}$. The abstract value $\lambda x_2.t_2$ represents all closures originating at λx_2, and `const` 1 represents intrinsic functions in C of arity 1 (in our example, $\neg$). The body of the abstract lambda is the variable name labeling the body, not the body itself. For example, t_2 labels the body `let` t_2 `=` *one* `in` t_2 of λx_2. Due to ANF, all terms have a label, which the function NAME in Algorithm 1 formalizes.

We also define booleans $\mathit{unaligned}_x$ that state whether or not the expression labeled by x is unaligned. For example, we previously reasoned that $\mathit{unaligned}_x =$ true for $x \in \{t_2, t_3, t_4, t_5\}$ in Fig. 4. The alignment analysis aims to determine *minimal* sets S_x and boolean assignments of $\mathit{unaligned}_x$ for every program variable $x \in X$. A trivial solution is that all abstract values (there is a finite number of them in the program) flow to each program variable and that $\mathit{unaligned}_x =$ true for all $x \in X$. This solution is sound but useless. To compute a more precise solution, we follow the rules given by *constraints* $\mathbf{c} \in R$.[†]

We present the constraints through the GENERATECONSTRAINTS function in Algorithm 1 and for the example in Fig. 4. There are no constraints for variables that occur at the end of ANF `let` sequences (line 2 in Algorithm 1), and the case for `let` expressions (lines 3–36) instead produces all constraints. The cases for aliases (line 6), intrinsics (line 7), `assume` (line 35), and `weight` (line 36) are the most simple. Aliases of the form `let` x `=` y `in` $\mathbf{t}_2$ establish $S_y \subseteq S_x$. That is, all abstract values at y are also in x. Intrinsic operations results in a `const` abstract value. For example, the definition of n_1 at line 1 in Fig. 4 results in the constraint `const` $1 \in S_{n_1}$. Applications of `assume` are the source of stochastic values. For example, the definition of a_1 at line 10 results in the constraint `stoch` $\in S_{a_1}$. Note that `assume` cannot produce any other abstract values, as we only

Algorithm 1 Constraint generation function for $\mathbf{t} \in T_{\text{ANF}}$. We denote the power set of a set E with $\mathcal{P}(E)$.

```
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 $\{\texttt{const}\ |c| \in S_x\}$
 8            else $\varnothing$
 9      | $\lambda y.\ \mathbf{t}_y \to$ GENERATECONSTRAINTS($\mathbf{t}_y$)
10        $\cup\ \{\lambda y.\ \text{NAME}(\mathbf{t}_y) \in S_x\}$
11        $\cup\ \{unaligned_y \Rightarrow unaligned_n$
12            $\mid n \in \text{NAMES}(t_y)\}$
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          $\forall n\ (\texttt{const}\ n \in S_{lhs}) \wedge (n > 1)$
17            $\Rightarrow \texttt{const}\ n - 1 \in S_x$,
18          $\texttt{stoch} \in S_{lhs} \Rightarrow \texttt{stoch} \in S_x$,
19          $\texttt{const}\ \_ \in S_{lhs}$
20            $\Rightarrow (\texttt{stoch} \in S_{rhs} \Rightarrow \texttt{stoch} \in S_x)$,
21          $unaligned_x$
22            $\Rightarrow (\forall y\ \lambda y.\_ \in S_{lhs} \Rightarrow unaligned_y)$,
23          $\texttt{stoch} \in S_{lhs}$
24            $\Rightarrow (\forall y\ \lambda y.\_ \in S_{lhs} \Rightarrow unaligned_y)$
25        $\}$
26    | if $y$ then $\mathbf{t}_t$ else $\mathbf{t}_e \to$
27      GENERATECONSTRAINTS($\mathbf{t}_t$)
28        $\cup$ GENERATECONSTRAINTS($\mathbf{t}_e$)
29        $\cup\ \{S_{\text{NAME}(\mathbf{t}_t)} \subseteq S_x, S_{\text{NAME}(\mathbf{t}_e)} \subseteq S_x$,
30            $\texttt{stoch} \in S_y \Rightarrow \texttt{stoch} \in S_x\}$
31        $\cup\ \{unaligned_x \Rightarrow unaligned_n$
32            $\mid n \in \text{NAMES}(\mathbf{t}_t) \cup \text{NAMES}(\mathbf{t}_e)\}$
33        $\cup\ \{\texttt{stoch} \in S_y \Rightarrow unaligned_n$
34            $\mid n \in \text{NAMES}(\mathbf{t}_t) \cup \text{NAMES}(\mathbf{t}_e)\}$
35    | assume _ $\to \{\texttt{stoch} \in S_x\}$
36    | weight _ $\to \varnothing$
37
38  function NAME(t): $T_{\text{ANF}} \to X$ =
39    match t with
40    | $x \to x$
41    | let $x = \mathbf{t}_1$ in $\mathbf{t}_2 \to$ NAME($\mathbf{t}_2$)
42
43  function NAMES(t): $T_{\text{ANF}} \to \mathcal{P}(X)$ =
44    match t with
45    | $x \to \varnothing$
46    | let $x$ = _ in $\mathbf{t}_2 \to \{x\} \cup$ NAMES($\mathbf{t}_2$)
47
48
49
50
```

allow distributions over intrinsics with arity 0 (see Definition 3). Finally, we use `weight` only for its side effect (likelihood updating), and therefore `weight`s do not produce any abstract values and consequently no constraints.

The cases for abstractions (line 9), applications (line 13), and `if`s (line 26) are more complex. The abstraction at line 4 in Fig. 4 generates (omitting the recursively generated constraints for the abstraction body $\mathbf{t}_y$) the constraints $\{\lambda x_1.x_1 \in S_{f_1}\} \cup \{unaligned_{x_1} \Rightarrow unaligned_{t_1}\}$. The first constraint is standard: the abstract lambda $\lambda x_1.x_1$ flows to S_{f_1}. The second constraint states that if the abstraction is unaligned, all expressions in its body (here, only t_1) are unaligned. We define the sets of expressions within abstraction bodies and `if` branches through the NAMES function in Algorithm 1 (line 43).

The application f_5 *one* at line 18 in Fig. 4 generates the constraints

$$\begin{aligned}
\{&\forall z \forall y\ \ \lambda z.y \in S_{f_5} \Rightarrow (S_{one} \subseteq S_z) \wedge (S_y \subseteq S_{v_4}),\\
&\forall n\ \ (\texttt{const}\ \ n \in S_{f_5}) \wedge (n > 1) \Rightarrow \texttt{const}\ \ n - 1 \in S_{v_4},\\
&\texttt{stoch} \in S_{f_5} \Rightarrow \texttt{stoch} \in S_{v_4},\\
&\texttt{const}\ \ \_ \in S_{f_5} \Rightarrow (\texttt{stoch} \in S_{one} \Rightarrow \texttt{stoch} \in S_{v_4}),\\
&unaligned_{v_4} \Rightarrow (\forall y\ \ \lambda y.\_ \in S_{f_5} \Rightarrow unaligned_y),\\
&\texttt{stoch} \in S_{f_5} \Rightarrow (\forall y\ \ \lambda y.\_ \in S_{lhs} \Rightarrow unaligned_y)\}
\end{aligned} \tag{3}$$

The first constraint is standard: if an abstract value $\lambda z.y$ flows to f_5, the abstract values of *one* (the right-hand side) flow to z. Furthermore, the result of the application, given by the body name y, must flow to the result v_4 of the application.

The second constraint is also relatively standard: if an intrinsic function of arity n is applied, it produces a const of arity $n-1$. The other constraints are new and specific for stochastic values and unalignment. The third constraint states that if the function is stochastic, the result is stochastic. The fourth constraint states that if we apply an intrinsic function to a stochastic argument, the result is stochastic. We could also make the analysis of intrinsic applications less conservative through intrinsic-specific constraints. The fifth and sixth constraints state that if the expression (labeled by v_4) is unaligned or the function is stochastic, all abstract lambdas that flow to the function are unaligned.

The `if` resulting in f_5 at line 14 in Fig. 4 generates (omitting the recursively generated constraints for the branches $\mathbf{t}_t$ and $\mathbf{t}_e$) the constraints

$$\begin{gathered}\{S_{\text{NAME}(f_2)} \subseteq S_{f_5}, S_{\text{NAME}(f_3)} \subseteq S_{f_5}, \texttt{stoch} \in S_{a_1} \Rightarrow \texttt{stoch} \in S_{f_5}\} \\ \cup \{\mathit{unaligned}_{f_5} \Rightarrow \mathit{unaligned}_{t_5}\} \cup \{\texttt{stoch} \in S_{a_1} \Rightarrow \mathit{unaligned}_{t_5}\}\end{gathered} \tag{4}$$

The first two constraints are standard and state that the result of the branches flows to the result of the `if` expression. The remaining constraints are new. The third constraint states that if the condition is stochastic, the result is stochastic. The last two constraints state that if the `if` is unaligned or if the condition is stochastic, all names in the branches (here, only t_5) are unaligned.

Given constraints for a program, we need to compute a solution satisfying all constraints. We do this by repeatedly iterating through all the constraints and propagating abstract values accordingly. We terminate when we reach a fixed point, i.e., when no constraint results in an update of either S_x or $\mathit{unaligned}_x$ for any x in the program. We extend the 0-CFA constraint propagation algorithm to also handle the constraints generated for tracking stochastic values and unalignment.† Specifically, the algorithm is a function ANALYZEALIGN: $T_{\text{ANF}} \rightarrow ((X \rightarrow \mathcal{P}(A)) \times \mathcal{P}(X))$ that returns a map associating each variable to a set of abstract values and a set of unaligned variables. In other words, ANALYZEALIGN computes a solution to S_x and $\mathit{unaligned}_x$ for each x in the analyzed program. For example, ANALYZEALIGN($\mathbf{t}_{example}$) results in

$$\begin{gathered}S_{n_1} = \{\texttt{const 1}\} \;\; S_{n_2} = \{\texttt{const 1}\} \;\; S_{f_1} = \{\lambda x_1.x_1\} \;\; S_{f_2} = \{\lambda x_2.t_2\} \\ S_{f_3} = \{\lambda x_3.t_3\} \;\; S_{f_4} = \{\lambda x_4.t_4\} \;\; S_{a_1} = \{\texttt{stoch}\} \;\; S_{v_2} = \{\texttt{stoch}\} \\ S_{f_5} = \{\lambda x_2.t_2, \lambda x_3.t_3, \texttt{stoch}\} \;\; S_{v_4} = \{\texttt{stoch}\} \;\; S_n = \varnothing \mid \text{other } n \in X \\ \mathit{unaligned}_n = \text{true} \mid n \in \{t_2, t_3, t_4, t_5\} \;\; \mathit{unaligned}_n = \text{false} \mid \text{other } n \in X.\end{gathered} \tag{5}$$

The example confirms our earlier intuition: an intrinsic ($\neg$) flows to n_1, `stoch` flows to a_1, f_5 is stochastic and originates at either $(\lambda x_2.t_2)$ or $(\lambda x_3.t_3)$, and the unaligned variables are t_2, t_3, t_4, and t_5. We now give soundness results.

Lemma 1 (0-CFA soundness). *For every* $\mathbf{t} \in T_{\text{ANF}}$*, the solution produced by* ANALYZEALIGN($\mathbf{t}$) *satisfies the constraints* GENERATECONSTRAINTS($\mathbf{t}$).

Proof. The well-known soundness of 0-CFA extends to the new alignment constraints. See, e.g., Nielson et al. [34, Chapter 3] and Shivers [42]. □

Theorem 1 (Alignment analysis soundness). *Assume* $\mathbf{t} \in T_{\text{ANF}}$, $\mathcal{A}_{\mathbf{t}}$ *from Definition 6, and an assignment to* S_x *and* $\mathit{unaligned}_x$ *for* $x \in X$ *according to* $\text{ANALYZEALIGN}(\mathbf{t})$. *Let* $\widehat{A}_{\mathbf{t}} = \{x \mid \neg \mathit{unaligned}_x\}$ *and take arbitrary* $\varnothing \vdash \mathbf{t} \;{}^{s_1}{\Downarrow}^{w_1}_{l_1}\; \mathbf{v}_1$ *and* $\varnothing \vdash \mathbf{t} \;{}^{s_2}{\Downarrow}^{w_2}_{l_2}\; \mathbf{v}_2$. *Then,* $l_1|_{\widehat{A}_{\mathbf{t}}} = l_2|_{\widehat{A}_{\mathbf{t}}}$ *and consequently* $\widehat{A}_{\mathbf{t}} \subseteq A_{\mathbf{t}}$ *for at least one* $A_{\mathbf{t}} \in \mathcal{A}_{\mathbf{t}}$.

The proof[†] uses simultaneous structural induction over the derivations $\varnothing \vdash \mathbf{t} \;{}^{s_1}{\Downarrow}^{w_1}_{l_1}\; \mathbf{v}_1$ and $\varnothing \vdash \mathbf{t} \;{}^{s_2}{\Downarrow}^{w_2}_{l_2}\; \mathbf{v}_2$. At corresponding stochastic branches or stochastic function applications in the two derivations, a separate structural induction argument shows that, for the `let`-sequences l'_1 and l'_2 of the two stochastic subderivations, $l'_1|_{\widehat{A}_{\mathbf{t}}} = l'_2|_{\widehat{A}_{\mathbf{t}}} = []$. Combined, the two arguments give the result.

The result $\widehat{A}_{\mathbf{t}} \subseteq A_{\mathbf{t}}$ (cf. Definition 6) shows that the analysis is conservative.

4.3 Dynamic Alignment

An alternative to static alignment is *dynamic* alignment, which we explored in early stages when developing the alignment analysis. Dynamic alignment is fully context sensitive and amounts to introducing variables in programs that track (at runtime) when evaluation enters stochastic branching. To identify these stochastic branches, dynamic alignment also requires a runtime data structure that keeps track of the stochastic values. Similarly to k-CFA, dynamic alignment is potentially more precise than the 0-CFA approach. However, we discovered that dynamic alignment introduces significant runtime overhead. Again, we note that the models in Section 7 do not require a context-sensitive analysis, justifying the choice of 0-CFA over dynamic alignment and k-CFA.

5 Aligned SMC and MCMC

This section presents detailed algorithms for aligned SMC (Section 5.1) and aligned lightweight MCMC (Section 5.2). For a more pedagogical introduction to the algorithms, see Section 2. We assume a basic understanding of SMC and Metropolis–Hastings MCMC algorithms (see, e.g., Bishop [4]).

5.1 Aligned SMC

We saw in Section 2.1 that SMC operates by executing many instances of $\mathbf{t}$ concurrently, and resampling them at calls to `weight`. Critically, resampling requires that the inference algorithm can both suspend and resume executions. Here, we assume that we can create execution instances e of the probabilistic program $\mathbf{t}$, and that we can arbitrarily suspend and resume the instances. The technical details of suspension are beyond the scope of this paper. See Goodman and Stuhlmüller [14], Wood et al. [48], and Lundén et al. [25] for further details.

Algorithm 2 presents all steps for the aligned SMC inference algorithm. After running the alignment analysis and setting up the n execution instances, the algorithm iteratively executes and resamples the instances. Note that the algorithm resamples only at aligned weights (see Section 2.1).

Algorithm 2 Aligned SMC. The input is a program $\mathbf{t} \in T_{\text{ANF}}$ and the number of execution instances n.

```
1. Run the alignment analysis on $\mathbf{t}$, resulting in $\widehat{A}_{\mathbf{t}}$ (see Theorem 1).
2. Initiate $n$ execution instances $\{e_i \mid i \in \mathbb{N}, 1 \leq i \leq n\}$ of $\mathbf{t}$.
3. Execute all $e_i$ and suspend execution upon reaching an aligned weight (i.e., let $x$ = weight
   $w$ in t and $x \in \widehat{A}_{\mathbf{t}}$) or when the execution terminates naturally. The result is a new set of
   execution instances $e_i'$ with weights $w_i'$ accumulated from unaligned weights and the single final
   aligned weight during execution.
4. If all $e_i' = \mathbf{v}_i'$ (i.e., all executions have terminated and returned a value), terminate inference and
   return the set of weighted samples $(\mathbf{v}_i', w_i')$. The samples approximate the posterior probability
   distribution encoded by $\mathbf{t}$.
5. Resample the $e_i'$ according to their weights $w_i'$. The result is a new set of unweighted execution
   instances $e_i''$. Set $e_i \leftarrow e_i''$. Go to 3.
```

```
if assume Bernoulli(0.5) then
  weight 1; weight 10; true
else
  weight 10; weight 1; false
```

(a) Aligned better than unaligned.

```
if assume Bernoulli(0.1) then
  weight 9;
  if assume Bernoulli(0.5)
  then weight 1.5 else weight 0.5;
  true
else (weight 1; false)
```

(b) Unaligned better than aligned.

Fig. 5: Programs illustrating properties of aligned and unaligned SMC. Fig. (a) shows a program better suited for aligned SMC. Fig. (b) shows a program better suited for unaligned SMC.

We conjecture that aligned SMC is preferable over unaligned SMC for all practically relevant models, as the evaluation in Section 7 justifies. However, it is possible to construct contrived programs in which unaligned SMC has the advantage. Consider the programs in Fig. 5, both encoding Bernoulli(0.5) distributions in a contrived way using `weight`s. Fig. 5a takes one of two branches with equal probability. Unaligned SMC resamples at the first `weight`s in each branch, while aligned SMC does not because the branch is stochastic. Due to the difference in likelihood, many more `else` executions survive resampling compared to `then` executions. However, due to the final `weight`s in each branch, the branch likelihoods even out. That is, resampling at the first `weight`s is detrimental, and unaligned SMC performs worse than aligned SMC. Fig. 5b also takes one of two branches, but now with unequal probabilities. However, the two branches still have equal posterior probability due to the `weight`s. The nested if in the `then` branch does not modify the overall branch likelihood, but adds variance. Aligned SMC does not resample for any `weight` within the branches, as the branch is stochastic. Consequently, only 10% of the executions in aligned SMC take the `then` branch, while half of the executions take the `then` branch in unaligned SMC (after resampling at the first `weight`). Therefore, unaligned SMC better explores the `then` branch and reduces the variance due to the nested `if`, which results in overall better inference accuracy. We are not aware of any real model with the property in Fig. 5b. In practice, it seems best to always resample when using `weight` to condition on observed data. Such conditioning is, in practice, always done outside of stochastic branches, justifying the benefit of aligned SMC.

Algorithm 3 Aligned lightweight MCMC. The input is a program $\mathbf{t} \in T_{\text{ANF}}$, the number of steps n, and the global step probability $g > 0$.

```
1. Run the alignment analysis on $\mathbf{t}$, resulting in $\widehat{A}_{\mathbf{t}}$ (see Theorem 1).
2. Set $i \leftarrow 0$, $k \leftarrow 1$, and $l \leftarrow 1$. Call Run.
3. Set $i \leftarrow i + 1$. If $i = n$, terminate inference and return the samples $\{\mathbf{v}_j \mid j \in \mathbb{N}, 0 \leq j < n\}$.
   They approximate the probability distribution encoded by $\mathbf{t}$.
4. Uniformly draw an index $1 \leq j \leq |s_{i-1}|$ at random. Set $global \leftarrow$ true with probability $g$, and
   $global \leftarrow$ false otherwise. Set $w'_{-1} \leftarrow 1$, $w' \leftarrow 1$, $k \leftarrow 1$, $l \leftarrow 1$, and $reuse \leftarrow$ true. Call Run.
5. Compute the Metropolis–Hastings acceptance ratio $A = \min\left(1, \frac{w_i}{w_{i-1}} \frac{w'}{w'_{-1}}\right)$.
6. With probability $A$, accept $\mathbf{v}_i$ and go to 3. Otherwise, set $\mathbf{v}_i \leftarrow \mathbf{v}_{i-1}$, $w_i \leftarrow w_{i-1}$, $s_i \leftarrow s_{i-1}$,
   $p_i \leftarrow p_{i-1}$, $s'_i \leftarrow s'_{i-1}$, $p'_i \leftarrow p'_{i-1}$, and $n'_i \leftarrow n'_{i-1}$. Go to 3.

function Run() = Run $\mathbf{t}$ and do the following:
  – Record the total weight $w_i$ accumulated from calls to weight.
  – Record the final value $\mathbf{v}_i$.
  – At unaligned terms let $c$ = assume $d$ in $\mathbf{t}$ ($c \notin \widehat{A}_{\mathbf{t}}$), do the following.
      1. If $reuse$ = false, $global$ = true, $n'_{i-1,k,l} \neq c$, or if $s'_{i-1,k,l}$ does not exist, sample a
         value $x$ from $d$ and set $reuse \leftarrow$ false. Otherwise, reuse the sample $x = s'_{i-1,k,l}$ and set
         $w'_{-1} \leftarrow w'_{-1} \cdot p'_{i-1,k,l}$ and $w' \leftarrow w' \cdot f_d(c)$.
      2. Set $s'_{i,k,l} \leftarrow x$, $p'_{i,k,l} \leftarrow f_d(x)$, and $n'_{i,k,l} \leftarrow c$.
      3. Set $l \leftarrow l + 1$. In the program, bind $c$ to the value $x$ and resume execution.
  – At aligned terms let $c$ = assume $d$ in $\mathbf{t}$ ($c \in \widehat{A}_{\mathbf{t}}$), do the following.
      1. If $j = k$, $global$ = true, or if $s_{i-1,k}$ does not exist, sample a value $x$ from $d$ normally.
         Otherwise, reuse the sample $x = s_{i-1,k}$. Set $w'_{-1} \leftarrow w'_{-1} \cdot p_{i-1,k}$ and $w' \leftarrow w' \cdot f_d(x)$.
      2. Set $s_{i,k} \leftarrow x$ and $p_{i,k} \leftarrow f_d(x)$.
      3. Set $k \leftarrow k + 1$, $l \leftarrow 1$, and $reuse \leftarrow$ true. In the program, bind $c$ to the value $x$ and resume
         execution.
```

5.2 Aligned Lightweight MCMC

Aligned lightweight MCMC is a version of lightweight MCMC [47], where the alignment analysis provides information about how to reuse random draws between executions. Algorithm 3, a Metropolis–Hastings algorithm in the context of PPLs, presents the details. Essentially, the algorithm executes the program repeatedly using the Run function, and redraws one aligned random draw in each step, while reusing all other aligned draws and as many unaligned draws as possible (illustrated in Section 2.2). It is possible to formally derive the Metropolis–Hastings acceptance ratio in step 5.[†] A key property in Algorithm 3 due to alignment (Definition 6) is that the length of s_i (and p_i) is constant, as executing $\mathbf{t}$ always results in the same number of aligned random draws.

In addition to redrawing only one aligned random draw, each step has a probability $g > 0$ of being *global*—meaning that inference redraws *every* random draw in the program. Occasional global steps fix problems related to slow mixing and ergodicity of lightweight MCMC identified by Kiselyov [21]. In a global step, the Metropolis–Hastings acceptance ratio reduces to $A = \min\left(1, \frac{w_i}{w_{i-1}}\right)$.

6 Implementation

We implement the alignment analysis (Section 4), aligned SMC (Section 5.1), and aligned lightweight MCMC (Section 5.2) for the functional PPL *Miking*

CorePPL [25], implemented as part of the *Miking* framework [7]. We implement the alignment analysis as a core component in the Miking CorePPL compiler, and then use the analysis when compiling to two Miking CorePPL backends: RootPPL and Miking Core. RootPPL is a low-level PPL with built-in highly efficient SMC inference [25], and we extend the CorePPL to RootPPL compiler introduced by Lundén et al. [25] to support aligned SMC inference. Furthermore, we implement aligned lightweight MCMC inference standalone as a translation from Miking CorePPL to Miking Core. Miking Core is the general-purpose programming language of the Miking framework, currently compiling to OCaml.

The idealized calculus in (1) does not capture all features of Miking CorePPL. In particular, the alignment analysis implementation must support records, variants, sequences, and pattern matching over these. Extending 0-CFA to such language features is not new, but it does introduce a critical challenge for the alignment analysis: identifying all possible stochastic branches. Determining stochastic `if`s is straightforward, as we simply check if `stoch` flows to the condition. However, complications arise when we add a `match` construct (and, in general, any type of branching construct). Consider the extension

$$\begin{aligned}
&\mathbf{t} ::= \ \ldots \ \mid \ \texttt{match}\ \mathbf{t}\ \texttt{with}\ \mathbf{p}\ \texttt{then}\ \mathbf{t}\ \texttt{else}\ \mathbf{t} \ \mid \ \texttt{\{}k_1 = x_1\texttt{,}\ \ldots\texttt{,}\ k_n = x_n\texttt{\}} \\
&\mathbf{p} ::= \ x \ \mid \ \text{true} \ \mid \ \text{false} \ \mid \ \texttt{\{}k_1 = \mathbf{p}\texttt{,}\ \ldots\texttt{,}\ k_n = \mathbf{p}\texttt{\}} \\
&x, x_1, \ldots, x_n \in X \quad k_1, \ldots, k_n \in K \quad n \in \mathbb{N}
\end{aligned} \tag{6}$$

of (1), adding records and simple pattern matching. K is a set of record keys. Assume we also extend the abstract values as $\mathbf{a} ::= \ldots \ \mid \ \texttt{\{}k_1 = X_1, \ldots, k_n = X_n\texttt{\}}$, where $X_1, \ldots, X_n \subseteq X$. That is, we add an abstract record tracking the names in the program that flow to its entries. Consider the program `match` t_1 `with {` $a = x_1$`,` $b = \text{false}$ `} then` t_2 `else` t_3. This `match` is, similar to `if`s, stochastic if `stoch` $\in S_{t_1}$. It is also, however, stochastic in other cases. Assume we have two program variables, x and y, such that `stoch` $\in S_x$ and `stoch` $\notin S_y$. Now, the `match` is stochastic if, e.g., $\texttt{\{}a = \{y\}\texttt{,}\ b = \{x\}\texttt{\}} \in S_{t_1}$, because the random value flowing from x to the pattern false may not match because of randomness. However, it is *not* stochastic if, instead, $S_{t_1} = \{\texttt{\{}a = \{x\}\texttt{,}\ b = \{y\}\texttt{\}}\}$. The randomness of x does *not* influence whether or not the branch is stochastic—the variable pattern x_1 for label a always matches.

Our alignment analysis implementation handles the intricacies of identifying stochastic `match` cases for nested record, variant, and sequence patterns. In total, the alignment analysis, aligned SMC, and aligned lightweight MCMC implementations consist of approximately 1000 lines of code directly contributed as part of this paper. The code is available on GitHub [2].

7 Evaluation

This section evaluates aligned SMC and aligned lightweight MCMC on a set of models encoded in Miking CorePPL: CRBD [33,39] in Sections 7.1 and 7.5, ClaDS [28,39] in Section 7.2, state-space aircraft localization in Section 7.3, and latent Dirichlet allocation in Section 7.4. CRBD and ClaDS are non-trivial models of considerable interest in evolutionary biology and phylogenetics [39]. Similarly, LDA is a non-trivial topic model [5]. Running the alignment analysis took approximately 5 ms–30 ms for all models considered in the experiment, justifying that the time complexity is not a problem in practice.

We compare aligned SMC with standard unaligned SMC [14], which is identical to Algorithm 2, except that it resamples at *every* call to `weight`.† We carefully checked that automatic alignment corresponds to previous manual alignments of each model. For all SMC experiments, we estimate the *normalizing constant* produced as a by-product of SMC inference rather than the complete posterior distributions. The normalizing constant, also known as marginal likelihood or model evidence, frequently appears in Bayesian inference and gives the probability of the observed data averaged over the prior. The normalizing constant is useful for model comparison as it measures how well different probabilistic models fit the data (a larger normalizing constant indicates a better fit).

We ran aligned and unaligned SMC with Miking CorePPL and the RootPPL backend configured for a single-core (compiled with GCC 7.5.0). Lundén et al. [25] shows that the RootPPL backend is significantly more efficient than other state-of-the-art PPL SMC implementations. We ran aligned and unaligned SMC inference 300 times (and with 3 warmup runs) for each experiment for 10^4, 10^5, and 10^6 executions (also known as *particles* in SMC literature).

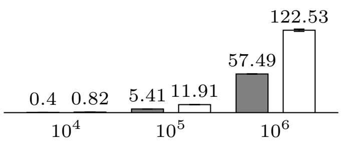
We compare aligned lightweight MCMC to lightweight MCMC.† We implement both versions as compilers from Miking CorePPL to Miking Core, which in turn compiles to OCaml (version 4.12). The lightweight MCMC databases are functional-style maps from the OCaml `Map` library. We set the global step probability to 0.1 for both aligned lightweight MCMC and lightweight MCMC. We ran aligned lightweight and lightweight MCMC inference 300 times for each experiment. We burned 10% of samples in all MCMC runs.

For all experiments, we used an Intel Xeon 656 Gold 6136 CPU (12 cores) and 64 GB of memory running Ubuntu 18.04.5.

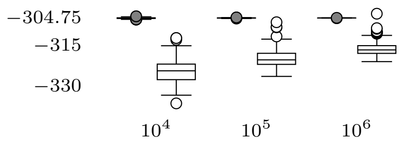
7.1 SMC: Constant Rate Birth-Death (CRBD)

This experiment considers the CRBD diversification model from [39] applied to the Alcedinidae phylogeny (Kingfisher birds, 54 extant species) [19]. We use fixed diversification rates to simplify the model, as unaligned SMC inference accuracy is too poor for the full model with priors over diversification rates. Aligned SMC is accurate for both the full and simplified models. The source code consists of 130 lines of code.† The total experiment execution time was 16 hours.

Fig. 6 presents the experiment results. Aligned SMC is roughly twice as fast and produces superior estimates of the normalizing constant. Unaligned SMC has not yet converged to the correct value -304.75 (available for this particular model due to the fixing the diversification rates) for 10^6 particles, while aligned SMC produces precise estimates already at 10^4 particles. Excess resampling is a significant factor in the increase in execution time for unaligned SMC, as each execution encounters far more resampling checkpoints than in aligned SMC.

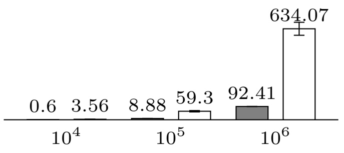


(a) Execution times.

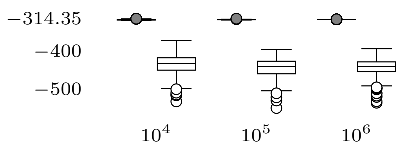


(b) Log normalizing constant estimates.

Fig. 6: SMC experiment results for CRBD. The x-axes give the number of particles. Fig. (a) shows execution times (in seconds) for aligned (gray) and unaligned (white) SMC. Error bars show one standard deviation. Fig. (b) shows box plot log normalizing constant estimates for aligned (gray) and unaligned (white) SMC. The analytically computed log normalizing constant is −304.75.



(a) Execution times.



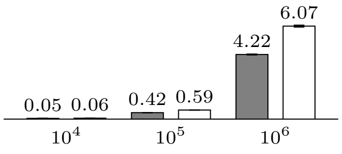
(b) Log normalizing constant estimates.

Fig. 7: SMC experiment results for ClaDS. The x-axes give the number of particles. Fig. (a) shows execution times (in seconds) for aligned (gray) and unaligned (white) SMC. Error bars show one standard deviation. Fig. (b) shows box plot log normalizing constant estimates for aligned (gray) and unaligned (white) SMC. The average estimate for aligned SMC with 10^6 particles is −314.35.

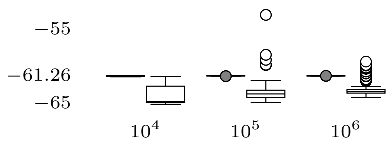
7.2 SMC: Cladogenetic Diversification Rate Shift (ClaDS)

A limitation of CRBD is that the diversification rates are constant. ClaDS [28,39] is a set of diversification models that allow *shifting* rates over phylogenies. We evaluate the ClaDS2 model for the Alcedinidae phylogeny. As in CRBD, we use fixed (initial) diversification rates to simplify the model on account of unaligned SMC. The source code consists of 147 lines of code.† Automatic alignment simplifies the ClaDS2 model significantly, as manual alignment requires collecting and passing weights around in unaligned parts of the program, which are later consumed by aligned `weight`s. The total experiment execution time was 67 hours.

Fig. 7 presents the experiment results. 12 unaligned runs for 10^6 particles and nine runs for 10^5 particles ran out of the preallocated stack memory for each particle (10 kB). We omit these runs from Fig. 7. The consequence of not aligning SMC is more severe than for CRBD. Aligned SMC is now almost seven times faster than unaligned SMC and the unaligned SMC normalizing constant estimates are significantly worse compared to the aligned SMC estimates. The unaligned SMC estimates do not even improve when moving from 10^4 to 10^6 particles (we need even more particles to see improvements). Again, aligned SMC produces precise estimates already at 10^4 particles.



(a) Execution times.



(b) Log normalizing constant estimates.

Fig. 8: SMC experiment results for the state-space aircraft localization model. The x-axes give the number of particles. Fig. (a) shows execution times (in seconds) for aligned (gray) and unaligned (white) SMC. Error bars show one standard deviation. Fig. (b) shows box plot log normalizing constant estimates on the y-axis for aligned (gray) and unaligned (white) SMC. The average estimate for aligned SMC with 10^6 particles is -61.26.

7.3 SMC: State-Space Aircraft Localization

This experiment considers an artificial but non-trivial state-space model for aircraft localization. The source code consists of 62 lines of code.† The total experiment execution time was 1 hour.

Fig. 8 presents the experiment results. The execution time difference is not as significant as for CRBD and ClaDS. However, the unaligned SMC normalizing constant estimates are again much less precise. Aligned SMC is accurate (centered at approximately -61.26) already at 10^4 particles. The model's straightforward control flow explains the less dramatic difference in execution time—there are at most ten unaligned likelihood updates in the aircraft model, while the number is, in theory, unbounded for CRBD and ClaDS. Therefore, the cost of extra resampling compared to aligned SMC is not as significant.

7.4 MCMC: Latent Dirichlet Allocation (LDA)

This experiment considers latent Dirichlet allocation (LDA), a topic model used in the evaluations by Wingate et al. [47] and Ritchie et al. [38]. We use a synthetic data set, comparable in size to the data set used by Ritchie et al. [38], with a vocabulary of 100 words, 10 topics, and 25 documents each containing 30 words. Note that we are not using methods based on collapsed Gibbs sampling [17], and the inference task is therefore computationally challenging even with a rather small number of words and documents. The source code consists of 31 lines of code.† The total experiment execution time was 41 hours.

The LDA model consists of only aligned random draws. As a consequence, aligned lightweight and lightweight MCMC reduces to the same inference algorithm, and we can compare the algorithms by just considering the execution times. The experiment also justifies the correctness of both algorithms.†

Fig. 9 presents the experiment results. Aligned lightweight MCMC is almost three times faster than lightweight MCMC. To justify the execution times with our implementations, we also implemented and ran the experiment with

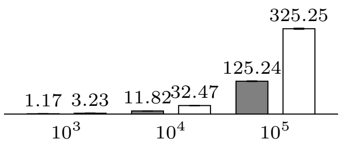


Fig. 9: MCMC experiment results for LDA showing execution time (in seconds) for aligned lightweight MCMC (gray) and lightweight MCMC (white). Error bars show one standard deviation and the x-axis the number of MCMC iterations.

lightweight MCMC in WebPPL [14] for 10^5 iterations, repeated 50 times (and with 3 warmup runs). The mean execution time was 383 s with standard deviation 5 s. We used WebPPL version 0.9.15 and Node version 16.18.0.

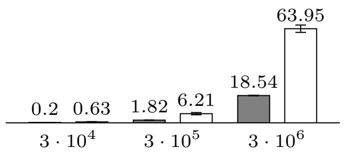
7.5 MCMC: Constant Rate Birth-Death (CRBD)

This experiment again considers CRBD. MCMC is not as suitable for CRBD as SMC, and therefore we use a simple synthetic phylogeny with six leaves and an age span of 5 age units (Alcedinidae used for the SMC experiment has 54 leaves and an age span of 35 age units). The source code for the complete model is the same as in Section 7.1, but we now allow the use of proper prior distributions for the diversification rates. The total experiment execution time was 7 hours.

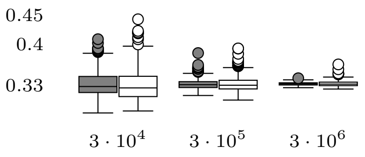
Unlike LDA, the CRBD model contains both unaligned and aligned random draws. Because of this, aligned lightweight MCMC and standard lightweight MCMC do *not* reduce to the same algorithm. To judge the difference in inference accuracy, we consider the mean estimates of the birth diversification rate produced by the two algorithms, in addition to execution times. The experiment results shows that the posterior distribution over the birth rate is unimodal†, which motivates using the posterior mean as a measure of accuracy.

Fig. 10 presents the experiment results. Aligned lightweight MCMC is approximately 3.5 times faster than lightweight MCMC. There is no obvious difference in accuracy. To justify the execution times and correctness of our implementations, we also implemented and ran the experiment with lightweight MCMC in WebPPL [14] for $3 \cdot 10^6$ iterations, repeated 50 times (and with 3 warmup runs). The mean estimates agreed with Fig. 10. The mean execution time was 37.1 s with standard deviation 0.8 s. The speedup compared to standard lightweight MCMC in Miking CorePPL is likely explained by the use of early termination in WebPPL, which benefits CRBD. Early termination easily combines with alignment but relies on execution suspension, which we do not currently use in our implementations. Note that aligned lightweight MCMC is faster than WebPPL even without early termination.

In conclusion, the experiments clearly demonstrate the need for alignment.



(a) Execution times.



(b) Birth rate mean estimates.

Fig. 10: MCMC experiment results for CRBD. The x-axes give the number of iterations. Fig. (a) shows execution times (in seconds) for aligned lightweight MCMC (gray) and lightweight MCMC (white). Error bars show one standard deviation. Fig. (b) shows box plot posterior mean estimates of the birth rate for aligned lightweight MCMC (gray) and lightweight MCMC (white). The average estimate for aligned lightweight MCMC with $3 \cdot 10^6$ iterations is 0.33.

8 Related Work

The approach by Wingate et al. [47] is closely related to ours. A key similarity with alignment is that executions reaching the same aligned checkpoint also have matching stack traces according to Wingate et al.'s addressing transform. However, Wingate et al. do not consider the separation between unaligned and aligned parts of the program, their approach is not static, and they do not generalize to other inference algorithms such as SMC.

Ronquist et al. [39], Turing [12], Anglican [48], Paige and Wood [36], and van de Meent et al. [46] consider the alignment problem. Manual alignment is critical for the models in Ronquist et al. [39] to make SMC inference tractable, which strongly motivates the automatic alignment approach. The documentation of Turing states that: "The `observe` statements [i.e., likelihood updates] should be arranged so that every possible run traverses all of them in exactly the same order. This is equivalent to demanding that they are not placed inside stochastic control flow" [1]. Turing does not include any automatic checks for this property. Anglican [48] checks, at runtime (resulting in overhead), that all SMC executions encounter the same number of likelihood updates, and thus resamples the same number of times. If not, Anglican reports an error: "some `observe` directives [i.e., likelihood updates] are not global". This error refers to the alignment problem, but the documentation does not explain it further. Probabilistic C, introduced by Paige and Wood [36], similarly assumes that the number of likelihood updates is the same in all executions. Van de Meent et al. [46] state, in reference to SMC: "Each breakpoint [i.e., checkpoint] needs to occur at an expression that is evaluated in every execution of a program". Again, they do not provide any formal definition of alignment nor an automatic solution to enforce it.

Lundén et al. [24] briefly mention the general problem of selecting optimal resampling locations in PPLs for SMC but do not consider the alignment problem in particular. They also acknowledge the overhead resulting from not all SMC executions resampling the same number of times, which alignment avoids.

The PPLs Birch [31], Pyro [3], and WebPPL [14] support SMC inference. Birch and Pyro enforce alignment for SMC as part of model construction. Note that this is only true for SMC in Pyro—other Pyro inference algorithms use other modeling approaches. The approaches in Birch and Pyro are sound but demand more of their users compared to the alignment approach. WebPPL does not consider alignment and resamples at all likelihood updates for SMC.

Ritchie et al. [38] and Nori et al. [35] present MCMC algorithms for probabilistic programs. Ritchie et al. [38] optimize lightweight MCMC by Wingate et al. [47] through execution suspensions and callsite caching. The optimizations are independent of and potentially combines well with aligned lightweight MCMC. Another MCMC optimization which potentially combines well with alignment is due to Nori et al. [35]. They use static analysis to propagate observations backwards in programs to improve inference.

Information flow analyses [40] may determine if particular parts of a program execute as a result of different program inputs. Specifically, if program input is random, such approaches have clear similarities to the alignment analysis.

Many other PPLs exist, such as Gen [10], Venture [29], Edward [44], Stan [8], and AugurV2 [18]. Gen, Venture, and Edward focus on simplifying the joint specification of a model and its inference to give users low-level control, and do not consider automatic alignment specifically. However, the incremental inference approach [9] in Gen does use the addressing approach by Wingate et al. [47]. Stan and AugurV2 have less expressive modeling languages to allow more powerful inference. Alignment is by construction due to the reduced expressiveness.

Borgström et al. [6], Staton et al. [43], Ścibior et al. [41], and Vákár et al. [45] treat semantics and correctness for PPLs, but do not consider alignment.

9 Conclusion

This paper gives, for the first time, a formal definition of alignment in PPLs. Furthermore, we introduce a static analysis technique and use it to align checkpoints in PPLs and apply it to SMC and MCMC inference. We formalize the alignment analysis, prove its correctness, and implement it in Miking CorePPL. We also implement aligned SMC and aligned lightweight MCMC, and evaluate the implementations on non-trivial CRBD and ClaDS models from phylogenetics, the LDA topic model, and a state-space model, demonstrating significant improvements compared to standard SMC and lightweight MCMC.

Acknowledgments We thank Lawrence Murray, Johannes Borgström, and Jan Kudlicka for early discussions on the alignment idea, and Viktor Senderov for implementing ClaDS in Miking CorePPL. We also thank the anonymous reviewers at ESOP for their valuable comments.

References

1. Turing.jl. https://turing.ml/dev/ (2022), accessed: 2022-02-24
2. Miking DPPL. https://github.com/miking-lang/miking-dppl (2023), accessed: 2023-01-02
3. 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)
4. Bishop, C.M.: Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag (2006)
5. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent Dirichlet allocation. Journal of Machine Learning Research **3**, 993–1022 (2003)
6. 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)
7. 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)
8. 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)
9. Cusumano-Towner, M., Bichsel, B., Gehr, T., Vechev, M., Mansinghka, V.K.: Incremental inference for probabilistic programs. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 571–585. Association for Computing Machinery, New York, NY, USA (2018)
10. 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)
11. 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, New York, NY, USA (1993)
12. Ge, H., Xu, K., Ghahramani, Z.: Turing: a language for flexible probabilistic inference. In: International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain. pp. 1682–1690 (2018)
13. 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)
14. Goodman, N.D., Stuhlmüller, A.: The design and implementation of probabilistic programming languages. http://dippl.org (2014), accessed: 2022-02-24
15. Goodman, N.D., Tenenbaum, J.B., Contributors, T.P.: Probabilistic Models of Cognition. http://probmods.org/v2 (2016), accessed: 2022-06-10

16. 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)
17. Griffiths, T.L., Steyvers, M.: Finding scientific topics. Proceedings of the National academy of Sciences **101**(suppl_1), 5228–5235 (2004)
18. Huang, D., Tristan, J.B., Morrisett, G.: Compiling markov chain monte carlo algorithms for probabilistic modeling. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 111–125. Association for Computing Machinery, New York, NY, USA (2017)
19. 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)
20. Kahn, G.: Natural semantics. In: Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science. pp. 22–39. Springer-Verlag, Berlin, Heidelberg (1987)
21. Kiselyov, O.: Problems of the lightweight implementation of probabilistic programming. In: Proceedings of Workshop on Probabilistic Programming Semantics (2016)
22. Kozen, D.: Semantics of probabilistic programs. Journal of Computer and System Sciences **22**(3), 328–350 (1981)
23. Lew, A., Agrawal, M., Sontag, D., Mansinghka, V.: PClean: Bayesian data cleaning at scale with domain-specific probabilistic programming. In: Proceedings of The 24th International Conference on Artificial Intelligence and Statistics. vol. 130, pp. 1927–1935. PMLR (2021)
24. 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, Cham (2021)
25. 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, Cham (2022)
26. Lundén, D., Caylak, G., Ronquist, F., Broman, D.: Artifact: Automatic alignment in higher-order probabilistic programming languages (Jan 2023). https://doi.org/10.5281/zenodo.7572555
27. Lundén, D., Caylak, G., Ronquist, F., Broman, D.: Automatic alignment in higher-order probabilistic programming languages. arXiv e-prints p. arXiv:2301.11664 (2023)
28. 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)
29. Mansinghka, V.K., Schaechtle, U., Handa, S., Radul, A., Chen, Y., Rinard, M.: Probabilistic programming with programmable inference. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 603–616. Association for Computing Machinery, New York, NY, USA (2018)
30. Midtgaard, J.: Control-flow analysis of functional programs. ACM Computing Surveys **44**(3) (2012)
31. Murray, L.M., Schön, T.B.: Automated learning with a probabilistic programming language: Birch. Annual Reviews in Control **46**, 29–43 (2018)
32. Naesseth, C., Lindsten, F., Schön, T.: Elements of Sequential Monte Carlo. Foundations and Trends in Machine Learning Series, Now Publishers (2019)

33. Nee, S.: Birth-death models in macroevolution. Annual Review of Ecology, Evolution, and Systematics **37**(1), 1–17 (2006)
34. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)
35. Nori, A., Hur, C.K., Rajamani, S., Samuel, S.: R2: An efficient MCMC sampler for probabilistic programs. Proceedings of the AAAI Conference on Artificial Intelligence **28**(1) (2014)
36. Paige, B., Wood, F.: A compilation target for probabilistic programming languages. In: Xing, E.P., Jebara, T. (eds.) Proceedings of the 31st International Conference on Machine Learning. vol. 32, pp. 1935–1943. PMLR, Bejing, China (22–24 Jun 2014)
37. Pierce, B.C.: Types and programming languages. MIT press (2002)
38. 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. PMLR, Cadiz, Spain (2016)
39. 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)
40. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003)
41. 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)
42. Shivers, O.G.: Control-flow analysis of higher-order languages or taming lambda. Carnegie Mellon University (1991)
43. Staton, S., Yang, H., Wood, F., Heunen, C., Kammar, O.: Semantics for probabilistic programming: Higher-order functions, continuous distributions, and soft constraints. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 525–534. Association for Computing Machinery (2016)
44. Tran, D., Hoffman, M.D., Saurous, R.A., Brevdo, E., Murphy, K., Blei, D.M.: Deep probabilistic programming. In: International Conference on Learning Representations (2017)
45. Vákár, M., Kammar, O., Staton, S.: A domain theory for statistical probabilistic programming. Proceedings of the ACM on Programming Languages **3**(POPL) (2019)
46. van de Meent, J.W., Paige, B., Yang, H., Wood, F.: An introduction to probabilistic programming. arXiv e-prints p. arXiv:1809.10756 (2018)
47. Wingate, D., Stuhlmueller, A., Goodman, N.: Lightweight implementations of probabilistic programming languages via transformational compilation. In: Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. vol. 15, pp. 770–778. PMLR (2011)
48. Wood, F., Meent, J.W., Mansinghka, V.: A new approach to probabilistic programming inference. In: Proceedings of the 17th International Conference on Artificial Intelligence and Statistics. vol. 33, pp. 1024–1032. PMLR (2014)

Correction to: Programming Languages and Systems

Thomas Wies

Correction to:
T. Wies (Ed.): ***Programming Languages and Systems*****,**
LNCS 13990, https://doi.org/10.1007/978-3-031-30044-8

In the originally published version of chapter 12, the order of the author names was not alphabetical as expected. This has been corrected.

In the originally published version of chapter 20, there was an error in Theorem 1. This has been corrected.

The updated original version of these chapters can be found at
https://doi.org/10.1007/978-3-031-30044-8_12
https://doi.org/10.1007/978-3-031-30044-8_20

T. Wies (Ed.): ESOP 2023, LNCS 13990, p. C1, 2023.
https://doi.org/10.1007/978-3-031-30044-8_21

Author Index

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 565–566, 2023.
https://doi.org/10.1007/978-3-031-30044-8