

**John P. Gallagher
Martin Sulzmann (Eds.)**

LNCS 10818

Functional and Logic Programming

**14th International Symposium, FLOPS 2018
Nagoya, Japan, May 9–11, 2018
Proceedings**

 **Springer**

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, Lancaster, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Friedemann Mattern

ETH Zurich, Zurich, Switzerland

John C. Mitchell

Stanford University, Stanford, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

C. Pandu Rangan

Indian Institute of Technology Madras, Chennai, India

Bernhard Steffen

TU Dortmund University, Dortmund, Germany

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbrücken, Germany

More information about this series at <http://www.springer.com/series/7407>

John P. Gallagher · Martin Sulzmann (Eds.)

Functional and Logic Programming

14th International Symposium, FLOPS 2018
Nagoya, Japan, May 9–11, 2018
Proceedings

Editors
John P. Gallagher
Roskilde University
Roskilde
Denmark

Martin Sulzmann
Karlsruhe University of Applied Sciences
Karlsruhe
Germany

and

IMDEA Software Institute
Madrid
Spain

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-90685-0 ISBN 978-3-319-90686-7 (eBook)
<https://doi.org/10.1007/978-3-319-90686-7>

Library of Congress Control Number: 2018941546

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

© Springer International Publishing AG, part of Springer Nature 2018

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

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

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

Printed on acid-free paper

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

Preface

This volume contains the proceedings of the 14th International Symposium on Functional and Logic Programming – FLOPS 2018 – held in Nagoya, Japan, May 9–11, 2018.

FLOPS brings together practitioners, researchers, and implementers of declarative programming, to discuss mutually interesting results and common problems: theoretical advances, their implementations in language systems and tools, and applications of these systems in practice. The scope includes all aspects of the design, semantics, theory, applications, implementations, and teaching of declarative programming. FLOPS specifically aims to promote cross-fertilization between theory and practice and among different styles of declarative programming.

The call for papers attracted 41 submissions. Each submission was reviewed by at least three reviewers, either members of the Program Committee (PC) or external referees. After careful and thorough discussions, the PC accepted 17 papers. The program also includes three invited talks by William Byrd, Cédric Fournet, and Zhenjiang Hu.

The award for best paper was made by the PC to Makoto Hamana for the paper “Polymorphic Computation Rules: Automated Confluence, Type Inference, and Instance Validation.”

We would like to thank all invited speakers and authors for their contributions. We are grateful to the PC and external reviewers for their hard work and the help of the EasyChair conference management system for making our work of organizing FLOPS 2018 much easier. We thank the general chair, Makoto Tatsuta, for his support throughout the process and taking on many administrative responsibilities. The local chair, Koji Nakazawa, and the local Organizing Committee did an excellent job in setting up the conference and making sure everything ran smoothly.

Finally, we would like to thank our sponsor, Japan Society for Software Science and Technology (JSSST), Special Interest Group on Programming and Programming Languages (SIG-PPL), for their continuing support. We acknowledge the cooperation of ACM SIGPLAN.

March 2018

John P. Gallagher
Martin Sulzmann

Organization

Program Committee

María Alpuente	Universitat Politècnica de València, Spain
Nikolaj Bjørner	Microsoft, USA
Joachim Breitner	University of Pennsylvania, USA
Michael Codish	Ben-Gurion University of the Negev, Israel
Carsten Fuhs	Birkbeck, University of London, UK
John P. Gallagher	Roskilde University, Denmark and IMDEA Software Institute, Spain
Maria Garcia De La Banda	Monash University, Australia
Jacques Garrigue	Nagoya University, Japan
Samir Genaim	Universidad Complutense de Madrid, Spain
Robert Glück	University of Copenhagen, Denmark
Siau Cheng Khoo	National University of Singapore, Singapore
Naoki Kobayashi	The University of Tokyo, Japan
Michael Leuschel	University of Düsseldorf, Germany
Kenny Zhuo Ming Lu	School of Information Technology, Nanyang Polytechnic, Singapore
Jan Midtgaard	University of Southern Denmark, Denmark
Jorge A. Navas	SRI International, USA
Atsushi Ohori	Tohoku University, Japan
Bruno C. D. S. Oliveira	The University of Hong Kong, SAR China
Andreas Rossberg	Google, Germany
Didier Rémy	Inria, France
Chungchieh Shan	Indiana University, Bloomington, USA
Martin Sulzmann	Karlsruhe University of Applied Sciences, Germany
Harald Søndergaard	The University of Melbourne, Australia
Kazunori Ueda	Waseda University, Japan
Meng Wang	University of Kent, UK

Additional Reviewers

Amadini, Roberto	García-Pérez, Álvaro
Braüner, Torben	Goldberg, Mayer
Chitil, Olaf	Iwami, Munehiro
Escobar, Santiago	Kaarsgaard, Robin
Felgenhauer, Bertram	Kirkeby, Maja
Filinski, Andrzej	Lo, Siaw Ling
Frank, Michael	Lucas, Salvador

Martin-Martin, Enrique
Muslimany, Morad
Rowe, Reuben
Saikawa, Takahumi
Sapiña, Julia
Schachte, Peter
Schneider-Kamp, Peter
Shani, Guy
Smallbone, Nick
Stadtmueller, Kai
Stuckey, Peter J.

Ta, Quang-Trung
Tsukada, Takeshi
Tsushima, Kanae
Vazou, Niki
Villanueva, Alicia
Wang, Yanlin
Xie, Ningning
Yang, Linus
Yang, Ye
Zhang, Haoyuan
Zhao, Jinxu

Abstracts of Invited Talks

Can Programming Be Liberated from Unidirectional Style?

Zhenjiang Hu

National Institute of Informatics, SOKENDAI, Japan
hu@nii.ac.jp

Abstract. Programs usually run unidirectional; computing output from input and output is not changeable. It is, however, becoming more and more important to develop programs whose output is subject to change. One typical example is data synchronization, where we may want to have a consistent schedule information by synchronizing calendars in different formats on various systems, make a smart watch by synchronizing its configuration with the environment, or achieve data interoperability by synchronization data among subsystems. This situation imposes difficulty in using the current unidirectional programming languages to construct such synchronization programs, because it would require us to develop and maintain several unidirectional programs that are tightly coupled and should be kept consistent with each other.

In this talk, I will start by briefly reviewing the current work on bidirectional programming, a new programming paradigm for developing well-behaved bidirectional programs in order to solve various synchronization problems. I will then discuss the essence of bidirectional programming, talk about our recent progress on putback-based bidirectional programming, and show a framework for supporting systematical development of bidirectional programs. Finally, I will highlight its potential application to lay the software foundations for controlling, integrating, and coordinating decentralized data.

miniKanren: A Family of Languages for Relational Programming

William E. Byrd

Department of Computer Science and Hugh Kaul Precision Medicine Institute,
University of Alabama at Birmingham, USA
webyrd@uab.edu

Abstract. miniKanren is a family of constraint logic programming languages designed for exploring *relational programming*. That is, every program written in miniKanren represents a mathematical relation, and every argument to that program can be a fresh logic variable, a ground term, or a partially ground term containing logic variables. The miniKanren language and implementation has been carefully designed to support this relational style of programming—for example, miniKanren uses a complete, biased, interleaving search by default, and unification always uses the occurs check.

miniKanren provides constraints useful for writing interpreters, type inferencers, and parsers as relations. One interesting class of miniKanren programs are interpreters written for a Turing-complete subset of Lisp, supporting lists, symbols, mutual recursion, and higher-order functions. Since these interpreters are written as relations, they can perform advanced tasks such as example-based program synthesis “for free.” By taking advantage of the declarative properties of miniKanren, and the semantics of Lisp, we have been able to speed up some synthesis problems by 9 orders of magnitude with respect to the default miniKanren search. We are actively exploring how to use machine learning and neural networks to further improve synthesis search.

miniKanren has also been used to prototype language semantics, similarly to semantics engineering tools like PLT Redex. One variant of miniKanren— α Kanren, inspired by α Prolog—supports nominal unification, and can be used to implement capture-avoiding substitution as a relation. miniKanren’s relational nature makes creating an executable semantics for a language easy in some ways, frustrating in others.

The most recent use of miniKanren is as the foundation of mediKanren, a language and system for reasoning over biomedical data sources, as part of the National Institutes of Health’s National Center For Advancing Translational Sciences (NCATS) Data Translator project. We have scaled miniKanren to reason over SemMedDB, a database of 97 million biomedical facts, and are integrating other data sources into the mediKanren system.

Finally, miniKanren is designed to be easy to understand, teach, implement, and hack. Implementing the miniKanren core language, *microKanren*, has become a standard part of learning miniKanren; as a result, there are hundreds of implementations of miniKanren, embedded in dozens of host languages. We have invested great effort in writing accessible books and papers on miniKanren, giving talks at industry and academic conferences, and teaching summer schools

and tutorials. One interesting result of these efforts is that we have developed a loose, distributed group of miniKanren researchers around the world.

In my talk I will explore these aspects of miniKanren, describe the lessons we have learned over the past 15 years, and outline the directions for future work.

Building Verified Cryptographic Components Using F*

Cédric Fournet

Microsoft Research
fournet@microsoft.com

Abstract. The HTTPS ecosystem includes communications protocols such as TLS and QUIC, the X.509 public key infrastructure, and various supporting cryptographic algorithms and constructions. This ecosystem remains surprisingly brittle, with practical attacks and emergency patches many times a year. To improve their security, we are developing high-performance, standards-compliant, formally verified implementation of these components. We aim for our verified components to be drop-in replacements suitable for use in mainstream web browsers, servers, and other popular tools.

In this talk, I will give an overview of our approach and our results so far. I will present our verification toolchain, based on F*: a programming language with dependent types, programmable monadic effects, support for both SMT-based and interactive proofs, and extraction to C and assembly code. I will also illustrate its application using security examples, ranging from the functional correctness of optimized implementations of cryptographic algorithms to the security of (fragments of) the new TLS 1.3 Internet Standard.

See <https://fstar-lang.org/> for an online tutorial and research papers on F*, and <https://project-everest.github.io/> for its security applications to cryptographic libraries, TLS, and QUIC.

Contents

Formal Verification of the Correspondence Between Call-by-Need and Call-by-Name	1
<i>Masayuki Mizuno and Eijiro Sumii</i>	
Direct Encodings of NP-Complete Problems into Horn Sequents of Multiplicative Linear Logic	17
<i>Satoshi Matsuoka</i>	
λ to SKI, Semantically: Declarative Pearl.	33
<i>Oleg Kiselyov</i>	
Program Extraction for Mutable Arrays	51
<i>Kazuhiko Sakaguchi</i>	
Functional Pearl: Folding Polynomials of Polynomials.	68
<i>Chen-Mou Cheng, Ruey-Lin Hsu, and Shin-Cheng Mu</i>	
A Functional Perspective on Machine Learning via Programmable Induction and Abduction	84
<i>Steven Cheung, Victor Darvari, Dan R. Ghica, Koko Muroya, and Reuben N. S. Rowe</i>	
Polymorphic Rewrite Rules: Confluence, Type Inference, and Instance Validation	99
<i>Makoto Hamana</i>	
Confluence Modulo Equivalence with Invariants in Constraint Handling Rules.	116
<i>Daniel Gall and Thom Frühwirth</i>	
On Probabilistic Term Rewriting.	132
<i>Martin Avanzini, Ugo Dal Lago, and Akihisa Yamada</i>	
Equivalence Checking of Non-deterministic Operations	149
<i>Sergio Antoy and Michael Hanus</i>	
Optimizing Declarative Parallel Distributed Graph Processing by Using Constraint Solvers.	166
<i>Akimasa Morihata, Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Hideya Iwasaki</i>	

Breaking Symmetries with Lex Implications	182
<i>Michael Codish, Thorsten Ehlers, Graeme Gange, Avraham Itzhakov, and Peter J. Stuckey</i>	
Model Checking Parameterized by the Semantics in Maude	198
<i>Adrián Riesco</i>	
Automated Amortised Resource Analysis for Term Rewrite Systems	214
<i>Georg Moser and Manuel Schneckenreither</i>	
A Common Framework Using Expected Types for Several Type Debugging Approaches	230
<i>Kanae Tsushima and Olaf Chitil</i>	
CauDEr: A Causal-Consistent Reversible Debugger for Erlang	247
<i>Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal</i>	
Cheap Remarks About Concurrent Programs	264
<i>Michael Walker and Colin Runciman</i>	
Author Index	281



Formal Verification of the Correspondence Between Call-by-Need and Call-by-Name

Masayuki Mizuno^(✉) and Eijiro Sumii

Tohoku University, Sendai, Japan
mizuno@sf.ecei.tohoku.ac.jp

Abstract. We formalize the call-by-need evaluation of λ -calculus (with no recursive bindings) and prove its correspondence with call-by-name, using the Coq proof assistant.

It has been long argued that there is a gap between the high-level abstraction of non-strict languages—namely, *call-by-name* evaluation—and their actual *call-by-need* implementations. Although a number of proofs have been given to bridge this gap, they are not necessarily suitable for stringent, mechanized verification because of the use of a global heap, “graph-based” techniques, or “marked reduction”. Our technical contributions are twofold: (1) we give a simpler proof based on two forms of standardization, adopting de Bruijn indices for representation of (non-recursive) variable bindings along with Ariola and Felleisen’s small-step semantics, and (2) we devise a technique to significantly simplify the formalization by eliminating the notion of evaluation contexts—which have been considered essential for the call-by-need calculus—from the definitions.

1 Introduction

Background. The *call-by-name* evaluation strategy has been considered the high-level abstraction of non-strict functional languages since Abramsky [1] and Ong [20] adopted call-by-name evaluation to weak head-normal forms as a formalism of laziness. However, when it comes to implementations, call-by-name as it does not lead to efficient execution because function arguments are evaluated every time they are needed. Therefore, most implementations adopt the *call-by-need* strategy [27], that is: when a redex is found, it is saved in a freshly allocated memory region called a *thunk*; when the redex needs to be evaluated, the thunk is updated with the value of the redex for later reuse.

There has been a large amount of research to bridge the gap between call-by-name and call-by-need by proving their correspondence, that is,

if the call-by-need evaluation of a term results in a value, its call-by-name evaluation results in a corresponding value, and vice versa.

For example, Launchbury [16] defined natural (or “big-step”) semantics for call-by-need evaluation (with mutually recursive bindings) and proved its adequacy through denotational semantics. Ariola and Felleisen [4] and Maraist et al. [17] developed small-step call-by-need operational semantics, and proved their correspondence to call-by-name. Kesner [15] gave an alternative proof, based on normalization with non-idempotent intersection types, using Accattoli et al. [2]’s call-by-need semantics.

Existing formalisms and our contribution. In this paper, we mechanize a formalization of the call-by-need evaluation and its correspondence with call-by-name, using the Coq proof assistant.¹ To this goal, after careful design choices, we adapt Ariola and Felleisen’s small-step semantics, and give a simpler proof based on two forms of standardization.

In what follows, we review existing formalisms to explain our choices. Several abstract machines (e.g. [11, 14, 21]) have been proposed for call-by-need evaluation, but they are generally too low-level for formal verification of correspondence to call-by-name. Launchbury [16] defined call-by-need natural semantics $\Gamma : e \Downarrow \Delta : z$, meaning “term e under store Γ evaluates to value z together with the modified store Δ ”. Ariola and Felleisen [4] and Maraist et al. [17] independently defined small-step reduction based on let-bindings: $\mathbf{let } x = M \mathbf{ in } N$ represents term N with a thunk M pointed to by x . For example, in Ariola and Felleisen’s semantics, the term $(\lambda x.xx)((\lambda y.y)(\lambda z.z))$ is reduced as follows:

$$\begin{aligned}
& (\lambda x.xx)((\lambda y.y)(\lambda z.z)) \\
\rightarrow & \mathbf{let } x = (\lambda y.y)(\lambda z.z) \mathbf{ in } xx \\
& \quad - x \text{ is bound to } (\lambda y.y)(\lambda z.z) \\
\rightarrow & \mathbf{let } x = (\mathbf{let } y = \lambda z.z \mathbf{ in } y) \mathbf{ in } xx \\
& \quad - x \text{ is evaluated and } y \text{ is bound to } \lambda z.z \\
\rightarrow & \mathbf{let } x = (\mathbf{let } y = \lambda z.z \mathbf{ in } \lambda z.z) \mathbf{ in } xx \\
& \quad - y \text{ is substituted with its value} \\
\rightarrow & \underline{\mathbf{let } y = \lambda z.z \mathbf{ in } \mathbf{let } x = \lambda z.z \mathbf{ in } xx} \\
& \quad - \text{let-bindings are flattened} \\
\rightarrow & \dots
\end{aligned}$$

Note, in particular, that the underlined let-binding (of y) is moved forward outside the other let-binding (of x). Such “reassociation” of let-bindings is also required for reductions like $(\mathbf{let } x = \dots \mathbf{ in } \lambda y.x)z \rightarrow \mathbf{let } x = \dots \mathbf{ in } (\lambda y.x)z$.

Variable bindings have been a central topic in the theory of formal languages. Choice of a representation of bindings—such as de Bruijn indices [10], locally nameless representation [13, 18, 19], or (parametric) higher order abstract syntax (PHOAS) [8, 22]—is particularly crucial for formal definitions and proofs on proof assistants. We adopt de Bruijn indices along with the reduction in Ariola and Felleisen [4] and Maraist et al. [17] because of their relative simplicity when manipulating the bindings of variables to thunks. (By contrast, Launchbury’s natural semantics is based on a monotonically growing global heap and is still

¹ We believe that our approach can be adopted in other proof assistants as well.

low-level, requiring fresh name generation—a.k.a. “gensym”—for allocation of thunks.) An obstacle in formalizing their definitions is the *evaluation contexts*, which may insert multiple bindings at once and are harder to formalize with de Bruijn indices. We eliminate them by devising a predicate that defines when a term “needs” the value of a variable.

Although our modified *definition* of call-by-need reduction is suitable for formalization with de Bruijn indices, existing *proofs* are still hard to formalize. On one hand, the proof by Ariola and Felleisen [4] is based on informally introduced graph representation of terms for relating call-by-need and call-by-name reductions; as they themselves write, “a graph model for a higher-order language requires many auxiliary notions” [4, p. 3]. On the other hand, Maraist et al. [17]’s proof uses rather intricate “marks” on redexes and their reductions to prove the confluence of their non-deterministic reductions. We therefore devise a simpler proof, outlined as follows. As for the correspondence between terms during call-by-name and call-by-need reductions, we simply inline all let-bindings (denoted by M^{th}) which represent thunks in call-by-need. Then, roughly speaking, a call-by-need reduction step such as $\text{let } x = M \text{ in } N \rightarrow \text{let } x = M' \text{ in } N$ corresponds to multiple call-by-name steps like $N[x \mapsto M] \rightarrow_* N[x \mapsto M']$. We then appeal to Curry and Feys’ standardization theorem [5] as follows:

- For the “forward direction”, suppose that a term M is evaluated to an answer A by call-by-need. Then $M^{\text{th}} \xrightarrow{\beta}_* A^{\text{th}}$ by full β -reduction $\xrightarrow{\beta}$. By a corollary of standardization, there exists some call-by-name evaluation $M^{\text{th}} \xrightarrow{\text{name}}_* V$ with $V \xrightarrow{\beta}_* A^{\text{th}}$.
- The main challenge is to prove the converse direction. Suppose $M^{\text{th}} \xrightarrow{\text{name}}_* V$. We aim to prove M is evaluated by call-by-need to some answer A corresponding to V . By another corollary of standardization, M^{th} is terminating (regardless of non-determinism) by repetition of $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$. Since a call-by-need reduction step corresponds to $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$, the call-by-need evaluation must also terminate with some N . The correspondence between N and V is then proved via the forward direction above.

Our Coq script is available at: <https://github.com/fetburner/call-by-need>

Structure of the paper. We review the call-by-name λ -calculus in Sect. 2. Section 3 presents the syntax and semantics of Ariola and Felleisen’s call-by-need λ -calculus. Section 4 gives the outline of our new proof of the correspondence between call-by-need and call-by-name, based on standardization. Section 5 details our techniques for formalization in a proof assistant (Coq, to be specific). Section 6 discusses previous researches and their relationship to our approach. Section 7 concludes with future work.

2 λ -Calculus and Call-by-Name Evaluation

We define the syntax and basic β -reduction of λ -calculus as in Fig. 1. The expression $M[x \mapsto N]$ denotes capture-avoiding substitution of N for each free occurrence of x in M . The full β -reduction $\xrightarrow{\beta}$ is defined as the compatible closure of

Syntax

Terms	L, M, N	$::=$	$x \mid V \mid M N$
Values (weak head normal forms)	V	$::=$	$\lambda x.M$
Evaluation contexts	E_n	$::=$	$\square \mid E_n M$

Reduction rule

$$(\beta) \quad (\lambda x.M)N \rightarrow M[x \mapsto N]$$

Fig. 1. Definitions for the call-by-name λ -calculus

the reduction rule (β) . For any binary relation \xrightarrow{R} , we write the reflexive transitive closure of \xrightarrow{R} as \xrightarrow{R}_* . For example, the reflexive transitive closure of $\xrightarrow{\beta}$ is written $\xrightarrow{\beta}_*$.

The call-by-name reduction $\xrightarrow{\text{name}}$ is the closure of the base rule (β) by evaluation contexts E_n . For example, in call-by-name, $(\lambda x.xx)((\lambda y.y)(\lambda z.z))$ is reduced as follows:

$$\begin{aligned} & (\lambda x.xx)((\lambda y.y)(\lambda z.z)) \\ \xrightarrow{\text{name}} & (\lambda y.y)(\lambda z.z)((\lambda y.y)(\lambda z.z)) \\ \xrightarrow{\text{name}} & (\lambda z.z)((\lambda y.y)(\lambda z.z)) \\ \xrightarrow{\text{name}} & (\lambda y.y)(\lambda z.z) \\ \xrightarrow{\text{name}} & \lambda z.z \end{aligned}$$

The relation $\xrightarrow{\text{name}}$ is a partial function and included in $\xrightarrow{\beta}$. Note that, under full β -reduction, there exists a shorter reduction sequence:

$$(\lambda x.xx)((\lambda y.y)(\lambda z.z)) \xrightarrow{\beta} (\lambda x.xx)(\lambda z.z) \xrightarrow{\beta} (\lambda z.z)(\lambda z.z) \xrightarrow{\beta} \lambda z.z$$

In order to establish the correspondence with call-by-need evaluation, we will also focus on stuck states $E_n[x]$. The basic properties of call-by-name can then be summarized as follows:

Lemma 1 (basic properties of call-by-name evaluation).

1. $\xrightarrow{\text{name}}$ is a partial function.
2. If $E_n[x] = E'_n[y]$ then $x = y$.
3. For any term M , exactly one of the following holds:
 - (a) M is a value
 - (b) $M = E_n[x]$ for some E_n and x
 - (c) M is reducible by $\xrightarrow{\text{name}}$

Proof. Clause 1 and 2 follow from straightforward structural inductions on evaluation contexts (note that $\xrightarrow{\text{name}}$ is the closure of the base rule β by call-by-name evaluation contexts E_n), and clause 3 is proved by structural induction on term M . \square

Syntax

Values	V	$::=$	$\lambda x.M$
Answers	A	$::=$	$V \mid \mathbf{let} \ x = M \ \mathbf{in} \ A$
Terms	L, M, N	$::=$	$x \mid V \mid M \ N \mid \mathbf{let} \ x = M \ \mathbf{in} \ N$
Evaluation contexts	E	$::=$	$\square \mid E \ M \mid \mathbf{let} \ x = M \ \mathbf{in} \ E \mid \mathbf{let} \ x = E \ \mathbf{in} \ E'[x]$

Reduction rules

(I)	$(\lambda x.M)N \rightarrow \mathbf{let} \ x = N \ \mathbf{in} \ M$
(V)	$\mathbf{let} \ x = V \ \mathbf{in} \ E[x] \rightarrow \mathbf{let} \ x = V \ \mathbf{in} \ E[V]$
(C)	$(\mathbf{let} \ x = M \ \mathbf{in} \ A) \ N \rightarrow \mathbf{let} \ x = M \ \mathbf{in} \ A \ N$
(A)	$\mathbf{let} \ y = \mathbf{let} \ x = M \ \mathbf{in} \ A \ \mathbf{in} \ E[y] \rightarrow \mathbf{let} \ x = M \ \mathbf{in} \ \mathbf{let} \ y = A \ \mathbf{in} \ E[y]$

Fig. 2. Ariola and Felleisen's call-by-need λ -calculus

The following Lemma 2 (a call-by-name variant of the leftmost reduction theorem, which is a folklore) and Lemma 3 (a call-by-name variant of quasi-leftmost reduction theorem, which seems to be original) are corollaries of Curry and Feys' standardization theorem [5]. As outlined in the Introduction, they play a crucial role in proving the correspondence between call-by-name and call-by-need.

Lemma 2. *If $M \xrightarrow{\beta}_* V$, there exists V' such that $M \xrightarrow{\text{name}}_* V' \xrightarrow{\beta}_* V$.*

Lemma 3. *If $M \xrightarrow{\beta}_* V$ for some V , then M is terminating by $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$ (despite the non-determinism).*

As we shall see in Sect. 4, the auxiliary relation $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$ plays a key role when proving the correspondence between call-by-need and call-by-name reductions.

3 Ariola and Felleisen's Call-by-Need λ -Calculus

We show the syntax, reduction rules, and evaluation contexts of Ariola and Felleisen's calculus in Fig. 2.² By convention, we assume that, whenever we write $E[x]$ on paper, the variable x is not bound by E . Note that our formalization in Coq will use de Bruijn indices and does not need such a convention. The call-by-need reduction $\xrightarrow{\text{need}}$ is the closure of the base rules (I), (V), (C), and (A) by the evaluation contexts E . For example, as mentioned in the introduction, the term $(\lambda x.xx)((\lambda y.y)(\lambda z.z))$ is reduced as follows (the redexes are underlined):

² Strictly speaking, the reduction rules shown here are called *standard reduction rules* in their paper, as opposed to non-deterministic reduction. Note that the let-binding $\mathbf{let} \ x = M \ \mathbf{in} \ N$ is non-recursive.

$$\begin{array}{l}
\frac{(\lambda x.xx)((\lambda y.y)(\lambda z.z))}{\text{— by rule (I) under evaluation context } \square} \\
\frac{\text{need} \rightarrow \text{let } x = (\lambda y.y)(\lambda z.z) \text{ in } xx}{\text{— by (I) under let } x = \square \text{ in } xx} \\
\frac{\text{need} \rightarrow \text{let } x = (\text{let } y = \lambda z.z \text{ in } y) \text{ in } xx}{\text{— by (V) under let } x = \square \text{ in } xx} \\
\frac{\text{need} \rightarrow \text{let } x = (\text{let } y = \lambda z.z \text{ in } \lambda z.z) \text{ in } xx}{\text{— by (A) under } \square} \\
\frac{\text{need} \rightarrow \text{let } y = \lambda z.z \text{ in let } x = \lambda z.z \text{ in } xx}{\text{need} \rightarrow \dots}
\end{array}$$

Note that the value $\lambda z.z$ of x is shared between the two occurrences of x and is never computed twice. Note also that, although the above call-by-need reduction sequence may seem longer than necessary, the “administrative” reductions by (V), (C), and (A) do not contribute to “real” reductions. In order to distinguish administrative reductions when proving the correspondence between call-by-need and call-by-name evaluations, we also consider reductions limited to specific base rules as follows. The reduction $\xrightarrow{\text{VCA}}$ is the closure of the three base rules (V), (C), and (A) by evaluation contexts E . Similarly, the reduction $\xrightarrow{\text{I}}$ is defined by the closure of the base rule (I). Obviously, $\xrightarrow{\text{need}} = \xrightarrow{\text{I}} \cup \xrightarrow{\text{VCA}}$.

The points of Ariola and Felleisen’s semantics are twofold: the representation of sharing by the syntactic form **let**, and redex positions by evaluation contexts. Thanks to these techniques, their semantics is entirely syntactic (that is, with no need for heaps or denotational semantics), which is desirable for mechanized verification.

The above call-by-need reductions are defined so that they become deterministic:

Lemma 4 (determinacy of call-by-need reductions).

1. $\xrightarrow{\text{I}}$ is a partial function.
2. $\xrightarrow{\text{VCA}}$ is a partial function.
3. If $E[x] = E'[y]$, then $x = y$.
4. For any term M , exactly one of the following holds:
 - (a) M is an answer
 - (b) $M = E[x]$ for some E and x
 - (c) M is reducible by $\xrightarrow{\text{I}}$
 - (d) M is reducible by $\xrightarrow{\text{VCA}}$

Proof. Again by straightforward structural inductions (cf. Lemma 1). □

4 Outline of Our Standardization-Based Proof

Before presenting the Coq formalization, we outline our new proof of the correspondence between call-by-name and call-by-need evaluations, based on standardization.

The correspondence M^{th} of terms is defined by let expansion as follows:

Definition 1.

$$\begin{aligned}
x^\dagger &= x \\
(\lambda x.M)^\dagger &= \lambda x.M^\dagger \\
(M N)^\dagger &= M^\dagger N^\dagger \\
(\mathbf{let } x = M \mathbf{ in } N)^\dagger &= N^\dagger[x \mapsto M^\dagger]
\end{aligned}$$

Although the above definition is similar to Maraist et al. [17], they annotated terms and reductions with what they call “marks”, which they use to keep track of the inlined **let**-bindings, while we somehow “recover” their reduction by considering the auxiliary reduction relation $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$.

Lemma 5 (single-step correspondence).

1. $(M[x \mapsto N])^\dagger = M^\dagger[x \mapsto N^\dagger]$.
2. A^\dagger is a value for any answer A .
3. For any E and x , $E[x]^\dagger = E_n[x]$ for some E_n .
4. If $M \xrightarrow{\text{VCA}} N$ then $M^\dagger = N^\dagger$.
5. If $M \xrightarrow{\text{I}} N$ then $M^\dagger \xrightarrow{\text{name}} \circ \xrightarrow{\beta}_* N^\dagger$.

Note that call-by-name reduction of M^\dagger itself does not straightforwardly correspond to call-by-need reduction of M since the latter may reduce more redexes due to the sharing by **let**-bindings. For instance, the call-by-need evaluation

$$\begin{aligned}
& \mathbf{let } x = (\lambda y.y)(\lambda z.z) \mathbf{ in } x(\lambda w.x) \\
& \xrightarrow{\text{need}}_* \dots \mathbf{let } x = \lambda z.z \mathbf{ in } x(\lambda w.x) \\
& \xrightarrow{\text{need}} \dots \mathbf{let } x = \lambda z.z \mathbf{ in } (\lambda z.z)(\lambda w.x) \\
& \xrightarrow{\text{need}}_* \dots \mathbf{let } x = \lambda z.z \mathbf{ in } \dots (\lambda w.x)
\end{aligned}$$

(omitting irrelevant **let**-bindings) becomes

$$\begin{aligned}
& (\lambda y.y)(\lambda z.z)(\lambda w.(\lambda y.y)(\lambda z.z)) \\
& \xrightarrow{\text{name}} (\lambda z.z)(\lambda w.(\lambda y.y)(\lambda z.z)) \\
& \xrightarrow{\text{name}} \lambda w.(\lambda y.y)(\lambda z.z)
\end{aligned}$$

in call-by-name, leaving the β -redex $(\lambda y.y)(\lambda z.z)$ inside a λ -abstraction, which needs to be reduced by full β -reduction.

Proof (Lemma 5). The first two clauses are proved by obvious structural inductions, and the next three clauses follow from the structural inductions on evaluation contexts (note that $\xrightarrow{\text{I}}$ and $\xrightarrow{\text{VCA}}$ are the closure of base rules by evaluation contexts E). \square

We first consider the “soundness” direction of the correspondence, that is, any call-by-need evaluation has a corresponding call-by-name evaluation:

Theorem 1 (soundness). *If $M \xrightarrow{\text{need}}_* A$, then $M^\dagger \xrightarrow{\text{name}}_* V \xrightarrow{\beta}_* A^\dagger$ for some V .*

Proof. Suppose $M \xrightarrow{\text{need}}_* A$. Then $M^\dagger \xrightarrow{\beta}_* A^\dagger$ by clause 4 and 5 of Lemma 5, where A^\dagger is a value by clause 2 of Lemma 5. Then, by Lemma 2, we obtain the value V such that $M^\dagger \xrightarrow{\text{name}}_* V \xrightarrow{\beta}_* A^\dagger$. \square

The harder, converse direction (called “completeness”) is as follows:

Theorem 2 (completeness). *If $M^\dagger \xrightarrow{\text{name}}_* V$, then $M \xrightarrow{\text{need}}_* A$ and $V \xrightarrow{\beta}_* A^\dagger$ for some A .*

In addition to the fact that call-by-name reduction by itself is not “sufficient” for call-by-need as explained above, another problem is that termination under call-by-name does not immediately imply termination under call-by-need since administrative reductions in call-by-need become 0 step in call-by-name, as in clause 4 of Lemma 5. To address the latter issue, we show the termination of administrative reductions as follows:

Lemma 6. *Administrative reduction $\xrightarrow{\text{VCA}}$ is terminating.*

Proof. By the decrease of the following measure function $\| M \|_s$, indexed by environments s mapping **let**-bound variables to the measure of their right-hand sides (and defaulting to 1 for other variables).

$$\begin{aligned} \| x \|_s &= s(x) \\ \| \lambda x.M \|_s &= \| M \|_{s \circ [x \mapsto 1]} \\ \| M N \|_s &= 2 \| M \|_s + 2 \| N \|_s \\ \| \mathbf{let} \ x = M \ \mathbf{in} \ N \|_s &= 2 \| M \|_s + \| N \|_{s \circ [x \mapsto 1 + \| M \|_s]} \end{aligned}$$

\square

This proof is similar to Maraist et al. [17, p. 287] except for our treatment of variables based on environments.

We then prove the completeness theorem:

Proof. (Theorem 2). First, we show that the call-by-need reduction of M is normalizing. If there is an infinite call-by-need reduction sequence from M , then by Lemma 6 it must contain an infinite number of $\xrightarrow{1}$, and therefore by clause 5 of Lemma 5 there is an infinite reduction sequence consisting of $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$ from M^\dagger . However, this contradicts with Lemma 3 (since $M^\dagger \xrightarrow{\text{name}}_* V$ obviously implies $M^\dagger \xrightarrow{\beta}_* V$)³.

Given that M terminates in call-by-need, we next show its normal form N is an answer A . The reasoning is summarized in Fig. 3. By clause 4 of Lemma 4, if N is *not* an answer, it is *stuck* in call-by-need, that is, $N = E[x]$ for some E and x . Then, by clause 3 of Lemma 5, $E[x]^\dagger = E_n[x]$ for some E_n , that is N^\dagger is

³ Although this argument seems to be a proof by contradiction, our actual Coq proof is constructive, using an induction on the finite reduction sequence of $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$ from M^\dagger as we shall see in Sect. 5.

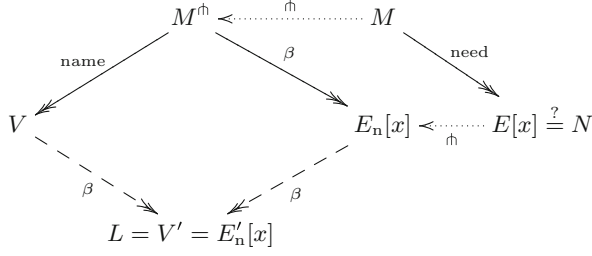


Fig. 3. Reasoning for non-stuckness of the normal form N

De Bruijn indexed syntax

Terms $L, M, N ::= x \mid V \mid M N$
 Values $V ::= \lambda.M$

Reduction rules

$$\frac{}{(\lambda.M)N \xrightarrow{\text{name}} M[0 \mapsto N]} \qquad \frac{}{(\lambda.M)N \xrightarrow{\beta} M[0 \mapsto N]}$$

Context rules

$$\frac{}{\text{needs}_n(x, x)} \qquad \frac{\text{needs}_n(M, x)}{\text{needs}_n(M N, x)} \qquad \frac{M \xrightarrow{\text{name}} M'}{M N \xrightarrow{\text{name}} M' N}$$

$$\frac{M \xrightarrow{\beta} M'}{M N \xrightarrow{\beta} M' N} \qquad \frac{N \xrightarrow{\beta} N'}{M N \xrightarrow{\beta} M N'} \qquad \frac{M \xrightarrow{\beta} M'}{\lambda.M \xrightarrow{\beta} \lambda.M'}$$

Fig. 4. Our modified definitions for the call-by-name λ -calculus and β -reduction

stuck in call-by-name. Also, by clause 4 and 5 of Lemma 5, $M^{\hat{h}} \xrightarrow{\beta}_* N^{\hat{h}}$. Then, by confluence of $\xrightarrow{\beta}$, there exists term L such that $N^{\hat{h}} \xrightarrow{\beta}_* L$ and $V \xrightarrow{\beta}_* L$, that is, L is a value and $N^{\hat{h}}$ reduces to it by $\xrightarrow{\beta}$. Since stuck states in call-by-name are preserved by $\xrightarrow{\beta}$, this contradicts with the fact that $N^{\hat{h}}$ is stuck in call-by-name.

Finally, we show $V \xrightarrow{\beta}_* A^{\hat{h}}$. By Theorem 1, we have some V' such that $M^{\hat{h}} \xrightarrow{\text{name}}_* V' \xrightarrow{\beta}_* A^{\hat{h}}$. We then obtain $V = V'$ by Lemma 1. \square

5 Formalization in Coq

The main points of our formalization in Coq are twofold: representation of binding by de Bruijn indices, and implicit treatment of evaluation contexts. We show the syntax and reduction rules of our modified call-by-name and call-by-need λ -calculi in Figs. 4 and 5.

De Bruijn indexed syntax

Values	V	$::=$	$\lambda.M$
Answers	A	$::=$	$V \mid \mathbf{let} _ = M \mathbf{in} A$
Terms	M, N	$::=$	$x \mid V \mid M N \mid \mathbf{let} _ = M \mathbf{in} N$

Reduction rules

$$\begin{array}{c}
\frac{}{(\lambda.M) N \xrightarrow{I} \mathbf{let} _ = N \mathbf{in} M} \qquad \frac{\mathbf{needs}(M, 0)}{\mathbf{let} _ = V \mathbf{in} M \xrightarrow{\text{VCA}} M[0 \mapsto V]} \\
\frac{}{(\mathbf{let} _ = M \mathbf{in} A) N \xrightarrow{\text{VCA}} \mathbf{let} _ = M \mathbf{in} A \uparrow_1 N} \\
\frac{\mathbf{needs}(N, 0)}{\mathbf{let} _ = (\mathbf{let} _ = M \mathbf{in} A) \mathbf{in} N \xrightarrow{\text{VCA}} \mathbf{let} _ = M \mathbf{in} \mathbf{let} _ = A \mathbf{in} \uparrow_1 N}
\end{array}$$

Context rules

$$\begin{array}{c}
\frac{}{\mathbf{needs}(x, x)} \qquad \frac{\mathbf{needs}(M, x)}{\mathbf{needs}(M N, x)} \qquad \frac{\mathbf{needs}(N, x+1)}{\mathbf{needs}(\mathbf{let} _ = M \mathbf{in} N, x)} \\
\frac{\mathbf{needs}(M, x) \quad \mathbf{needs}(N, 0)}{\mathbf{needs}(\mathbf{let} _ = M \mathbf{in} N, x)} \qquad \frac{N \xrightarrow{I} N'}{\mathbf{let} _ = M \mathbf{in} N \xrightarrow{I} \mathbf{let} _ = M \mathbf{in} N'} \\
\frac{M \xrightarrow{I} M'}{M N \xrightarrow{I} M' N} \qquad \frac{M \xrightarrow{I} M' \quad \mathbf{needs}(N, 0)}{\mathbf{let} _ = M \mathbf{in} N \xrightarrow{I} \mathbf{let} _ = M' \mathbf{in} N} \\
\frac{N \xrightarrow{\text{VCA}} N'}{\mathbf{let} _ = M \mathbf{in} N \xrightarrow{\text{VCA}} \mathbf{let} _ = M \mathbf{in} N'} \qquad \frac{M \xrightarrow{\text{VCA}} M'}{M N \xrightarrow{\text{VCA}} M' N} \\
\frac{M \xrightarrow{\text{VCA}} M' \quad \mathbf{needs}(N, 0)}{\mathbf{let} _ = M \mathbf{in} N \xrightarrow{\text{VCA}} \mathbf{let} _ = M' \mathbf{in} N}
\end{array}$$

Fig. 5. Our modified call-by-need λ -calculus

We use de Bruijn indices for simple manipulation of binding. Fortunately, Ariola and Felleisen’s semantics is straightforwardly adaptable for de Bruijn indices since only a constant number of bindings are inserted or hoisted by a reduction. We use the auxiliary operation $\uparrow_c M$ called “shifting”, which increments the indices of the free variables in M above the “cutoff” c as follows:

$$\begin{aligned} \uparrow_c x &= \begin{cases} x & \text{if } x < c \\ x + 1 & \text{if } x \geq c \end{cases} \\ \uparrow_c \lambda.M &= \lambda.\uparrow_{c+1}M \\ \uparrow_c(M N) &= (\uparrow_c M) (\uparrow_c N) \\ \uparrow_c(\mathbf{let } _ = M \mathbf{ in } N) &= (\mathbf{let } _ = \uparrow_c M \mathbf{ in } \uparrow_{c+1}N) \end{aligned}$$

We write $\uparrow M$ for $\uparrow_0 M$. In Coq, we use Autosubst [23] to automatically derive operations such as shifting on terms using de Bruijn indices, and their metatheories including basic properties of substitutions.

Although evaluation contexts reduce the number of reduction rules, explicit treatment of contexts often hinders automated reasoning.⁴ For example, consider the clause 4 of Lemma 4. To prove case (b) we need to find a concrete E such that $M = E[x]$, which requires second-order unification [12] in general. More concretely, in Coq, the lemma could be written like

```
Lemma answer_or_stuck_or_reducible M :
  answer M \ /
  (exists E x, evalctx E /\ M = E.[tvar x] /\ bv E <= x) \ /
  (exists E L N, evalctx E /\ M = E.[L] /\ reduceI L N) \ /
  (exists E L N, evalctx E /\ M = E.[L] /\ reduceVCA L N).
```

where $\text{bv } E \leq x$ means that x is not captured in E . This statement can be proved by induction on M , where we first encounter the case M is a variable x :

4 subgoals

```
x : var
=====
answer M \ /
(exists E y, evalctx E /\ tvar x = E.[tvar y] /\ bv E <= y) \ /
(exists E L N, evalctx E /\ tvar x = E.[L] /\ reduceI L N) \ /
(exists E L N, evalctx E /\ tvar x = E.[L] /\ reduceVCA L N)
```

However, automation fails even though the above disjunction is obviously true by the second clause with $E = []$:

Coq < eauto.

4 subgoals

```
x : var
=====
```

⁴ Another drawback is that evaluation contexts may introduce an arbitrary number of bindings and therefore need to be indexed by that number to coexist with de Bruijn indices, requiring heavy natural number calculations—like the Omega [9] library for Presburger arithmetic—in the mechanized proofs. Our approach will also obviate the need for such calculations.

```

answer M \ /
(exists E y, evalctx E /\ tvar x = E.[tvar y] /\ bv E <= y) \ /
(exists E L N, evalctx E /\ tvar x = E.[L] /\ reduceI L N) \ /
(exists E L N, evalctx E /\ tvar x = E.[L] /\ reduceVCA L N)

```

We avoid the above problem by eliminating evaluation contexts by expanding their definition in reductions $\xrightarrow{\beta}$, $\xrightarrow{\text{name}}$, \xrightarrow{I} , and $\xrightarrow{\text{VCA}}$, and devising *stuckness predicates* $\mathbf{needs}_n(M, x)$ and $\mathbf{needs}(M, x)$, corresponding to “ $M = E_n[x]$ for some E_n ” and “ $M = E[x]$ for some E ”, respectively, as in Fig. 5. (Note that, unlike evaluation contexts, each derivation rule of $\mathbf{needs}(M, x)$ inserts only at most one **let**-binding at once; cf. footnote 4.) The only deviation is thunk dereference **let** $x = V$ **in** $E[x]$ $\xrightarrow{\text{VCA}}$ **let** $x = V$ **in** $E[V]$, where we approximate the operation $E[V]$ by substitution $(E[x])[x \mapsto V]$. Although the latter may substitute extra occurrences of x in E itself, it is semantically equivalent to the former since V is already a value. Here our formalization favors simplicity over faithfulness and slightly differs from the original definition. It is also straightforward (though cumbersome) to adhere to the original by defining a partial function that substitutes a given value V with a given variable x in a redex position of a given term M .⁵

After the elimination of evaluation contexts, we can now prove clause 4 of Lemma 4 almost automatically as follows:

Lemma `answer_or_stuck_or_reducible M` :

```

answer M \ /
(exists x, needs M x) \ /
(exists N, reduceI M N) \ /
(exists N, reduceVCA M N).

```

Proof.

induction M as

```

[|? [Hanswer|[[[]|[[[]|[]]]]]
|]? [Hanswer|[[[]|[[[]|[]]]]] ? [|[[[]]|[[[]|[]]]]]; eauto 6;
inversion Hanswer; subst; eauto 6.

```

Qed.

Let us overview the other changes by our elimination of evaluation contexts: clause 2 of Lemma 1 becomes “if $\mathbf{needs}_n(M, x)$ and $\mathbf{needs}_n(M, y)$, then $x = y$ ”; case (b) of clause 3 of Lemma 1 changes to “ $\mathbf{needs}_n(M, x)$ for some x ”; clause 3 of Lemma 4 to “If $\mathbf{needs}(M, x)$ and $\mathbf{needs}(M, y)$, then $x = y$ ”; case (b) of clause 4 of Lemma 4 to “ $\mathbf{needs}(M, x)$ for some x ”; and clause 3 of Lemma 5 to “If $\mathbf{needs}(M, x)$ then $\mathbf{needs}_n(M^\dagger, x)$ ”. The proofs of these lemmas proceed by induction on the derivation of $\mathbf{needs}_n(M, x)$ or $\mathbf{needs}(M, x)$ instead of structural induction on evaluation contexts.

Another devisal in our Coq formalization is replacing the *reductio ad impossibile* for the normalization proof of Theorem 2 (completeness) in Sect. 4, with an intuitionistic, constructive proof as follows:

⁵ Indeed, we also formalized the original semantics and proved its correspondence to call-by-name. See: <https://github.com/fetburner/call-by-need>.

Proof. (Theorem 2, constructive version). By Lemma 3, M^\dagger is terminating by $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$. Let us define $L \Downarrow$, meaning that L is terminating by $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$, inductively as: $(\forall L'. L \xrightarrow{\text{name}} \circ \xrightarrow{\beta}_* L' \Rightarrow L' \Downarrow) \Rightarrow L \Downarrow$.⁶ We then prove a stronger property that, for any M' , $M' \Downarrow$ implies

for any M , if $M' = M^\dagger$, then M terminates by $\xrightarrow{\text{need}}$

by induction on the definition of $M' \Downarrow$. The above statement is trivially true if M is already a call-by-need normal form. If $M \xrightarrow{1} N$, then by clause 5 of Lemma 5 we have $M^\dagger \xrightarrow{\text{name}} \circ \xrightarrow{\beta}_* N^\dagger$, and by the induction hypothesis we have that N terminates by $\xrightarrow{\text{need}}$. If $M \xrightarrow{\text{VCA}} N$, then by clause 4 of Lemma 5 we have $M^\dagger = N^\dagger$ and the conclusion follows from a double, inner induction on reductions by $\xrightarrow{\text{VCA}}$, which is finite by Lemma 6.

The rest of the proof is similar to that in Sect. 4.

6 Related Work

Call-by-name and, to a lesser degree, call-by-need evaluations have been investigated for more than decades. We here focus on notable previous researches on the correspondence between call-by-need and call-by-name (other than Ariola and Felleisen [4], which we have already reviewed in Sect. 3), and discuss their differences from our approach.

- Launchbury [16] gave a natural semantics for call-by-need evaluation with mutually recursive bindings and proved its adequacy with respect to call-by-name evaluation. He defined judgements of the form $\Gamma : e \Downarrow \Delta : z$, meaning “term e under store Γ evaluates to value z , yielding a modified store Δ ”. The key of his semantics is a “dual use” of variables as pointers to thunks. This technique makes their semantics simpler than conventional operational semantics based on abstract machines.

However, Launchbury’s natural semantics is still challenging from the viewpoint of mechanical verification, not only because of mutual recursion, but also because of subtle variable convention: for example, let us evaluate term **let** $u = 3$, $f = (\lambda x. \mathbf{let} \ v = u + 1 \ \mathbf{in} \ v + x) \ \mathbf{in} \ f \ 2 + f \ 3$. We must replace the bound variable v in the body of the function f with some fresh variable v' every time f is called, because the pointers to the thunks are identified with variable names.

Recently, Breitner [6] formalized Launchbury’s natural semantics adopting nominal logic [25] *except for* mutually recursive heaps, which were represented by explicit names.

Vassena et al. [26] formalized Sestoft [24]’s small-step variant of Launchbury’s semantics, adopting de Bruijn indices. However, mutually recursive bindings are omitted from their language.

⁶ This definition is adopted from the accessibility predicate Acc in Coq.

- Maraist et al. [17] gave a small-step semantics for call-by-need evaluation and proved its correspondence (full abstraction) with call-by-name. Their semantics is almost the same as Ariola and Felleisen’s, making the former’s proof method also useful for the latter. (The difference between the two semantics is that, in Maraist et al., variables are values, and an additional reduction rule $\mathbf{let} \ x = M \ \mathbf{in} \ N \rightarrow N \quad (x \notin \mathbf{FV}(N))$ is introduced for garbage collection.)

A point of Maraist et al.’s proof is the introduction of *marks* on redexes for call-by-need reductions (I, V, C, or A). Their approach seems natural for proving the confluence of their non-deterministic reductions but significantly complicates the definitions of terms and reductions in a mechanized metatheory.

Although we did not directly adopt Maraist et al.’s formalism, it influenced our correspondence $M^{\#}$ of call-by-need terms with call-by-name, and our measure function in the proof of Lemma 6.

- Chang and Felleisen [7] proposed a variant of Ariola and Felleisen’s semantics and proved its correspondence with Launchbury’s semantics. They gave a simpler reduction rule with arguably more complicated evaluation contexts instead of administrative reductions. As argued in Sect. 5, we avoided any use of evaluation contexts for the sake of easier automation and formalization with de Bruijn indices.

7 Conclusion

We formalized a variant of Ariola and Felleisen’s small-step operational semantics of call-by-need λ -calculus and proved its correspondence with call-by-name, using the Coq proof assistant. For the formal verification, we developed a simpler proof based on two forms of standardization (Lemmas 2 and 3), adopting de Bruijn indices for representation of variable binding. Along the way, we simplified the formalization and enabled more automation by replacing evaluation contexts with more specific definitions (Sect. 5).

Future Work. We plan to extend our target language to more practical languages. Data types such as tuples and sums should be straightforward since they are already in weak head normal form. The most interesting challenge would be recursive definitions, because they cannot be completely inlined when establishing the correspondence with call-by-name (cf. Definition 1). A natural starting point here may be Ariola and Blom’s well-known theory of cyclic λ -calculus [3]. Another (though smaller) issue is binary operations (such as arithmetic addition +), for which non-deterministic (but strict) evaluation of the operands may be desirable.

Acknowledgments. We thank the anonymous reviewers for valuable comments and suggestions. This work was partially supported by JSPS KAKENHI Grant Number 15H02681 and 16K12409.

References

1. Abramsky, S.: The lazy lambda calculus. In: Turner, D.A. (ed.) *Research Topics in Functional Programming*, pp. 65–116. Addison-Wesley Publishing Co., Boston (1990)
2. Accattoli, B., Barenbaum, P., Mazza, D.: Distilling abstract machines. In: Jeuring, J., Chakravarty, M.M.T. (eds.) *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, Gothenburg, Sweden, 1–3 September 2014, pp. 363–376. ACM (2014)
3. Ariola, Z.M., Blom, S.: Cyclic lambda calculi. In: Abadi, M., Ito, T. (eds.) *TACS 1997*. LNCS, vol. 1281, pp. 77–106. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0014548>
4. Ariola, Z.M., Felleisen, M.: The call-by-need lambda calculus. *J. Funct. Program.* **7**(3), 265–301 (1997)
5. Barendregt, H.P.: *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, vol. 103, Revised edn. North-Holland, New York (1984)
6. Breitner, J.: The adequacy of Launchbury’s natural semantics for lazy evaluation. *J. Funct. Program.* **28**, e1 (2018)
7. Chang, S., Felleisen, M.: The call-by-need lambda calculus, revisited. In: Seidl, H. (ed.) *ESOP 2012*. LNCS, vol. 7211, pp. 128–147. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_7
8. Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: Hook, J., Thiemann, P. (eds.) *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2008*, Victoria, BC, Canada, 20–28 September 2008, pp. 143–156. ACM (2008)
9. Crégut, P.: Omega: a solver of quantifier-free problems in Presburger arithmetic. In: *The Coq Proof Assistant Reference Manual, Version 8.7.0* (2017)
10. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagation. Math. (Proc.)* **75**(5), 381–392 (1972)
11. Fairbairn, J., Wray, S.: Tim: a simple, lazy abstract machine to execute super-combinators. In: Kahn, G. (ed.) *FPCA 1987*. LNCS, vol. 274, pp. 34–45. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-18317-5_3
12. Goldfarb, W.D.: The undecidability of the second-order unification problem. *Theor. Comput. Sci.* **13**, 225–230 (1981)
13. Gordon, A.D.: A mechanisation of name-carrying syntax up to alpha-conversion. In: Joyce, J.J., Seger, C.-J.H. (eds.) *HUG 1993*. LNCS, vol. 780, pp. 413–425. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-57826-9_152
14. Johnsson, T.: Efficient compilation of lazy evaluation. In: Deusen, M.S.V., Graham, S.L. (eds.) *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, Montreal, Canada, 17–22 June 1984, pp. 58–69. ACM (1984)
15. Kesner, D.: Reasoning about call-by-need by means of types. In: Jacobs, B., Löding, C. (eds.) *FoSSaCS 2016*. LNCS, vol. 9634, pp. 424–441. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49630-5_25
16. Launchbury, J.: A natural semantics for lazy evaluation. In: Deusen, M.S.V., Lang, B. (eds.) *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, South Carolina, USA, January 1993, pp. 144–154. ACM Press (1993)

17. Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. *J. Funct. Program.* **8**(3), 275–317 (1998)
18. McBride, C., McKinna, J.: Functional pearl: I am not a number-I am a free variable. In: Nilsson, H. (ed.) *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004*, Snowbird, UT, USA, 22–22 September 2004, pp. 1–9. ACM (2004)
19. McKinna, J., Pollack, R.: Some lambda calculus and type theory formalized. *J. Autom. Reason.* **23**(3–4), 373–409 (1999)
20. Ong, C.L.: Fully abstract models of the lazy lambda calculus. In: *29th Annual Symposium on Foundations of Computer Science*, White Plains, New York, USA, 24–26 October 1988, pp. 368–376. IEEE Computer Society (1988)
21. Peyton Jones, S.L.: Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *J. Funct. Program.* **2**(2), 127–202 (1992)
22. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Wexelblat, R.L. (ed.) *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, USA, 22–24 June 1988, pp. 199–208. ACM (1988)
23. Schäfer, S., Tebbi, T., Smolka, G.: Autosubst: reasoning with de Bruijn terms and parallel substitutions. In: Urban, C., Zhang, X. (eds.) *ITP 2015*. LNCS, vol. 9236, pp. 359–374. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_24
24. Sestoft, P.: Deriving a lazy abstract machine. *J. Funct. Program.* **7**(3), 231–264 (1997)
25. Urban, C.: Nominal techniques in Isabelle/HOL. *J. Autom. Reason.* **40**(4), 327–356 (2008)
26. Vassena, M., Breitner, J., Russo, A.: Securing concurrent lazy programs against information leakage. In: *30th IEEE Computer Security Foundations Symposium, CSF 2017*, Santa Barbara, CA, USA, 21–25 August 2017, pp. 37–52 (2017)
27. Wadsworth, C.P.: *Semantics and pragmatics of the lambda calculus*. Ph.D. thesis, Oxford University (1971)



Direct Encodings of NP-Complete Problems into Horn Sequents of Multiplicative Linear Logic

Satoshi Matsuoka^(✉)

National Institute of Advanced Industrial Science and Technology (AIST),
1-1-1 Umezono, Tsukuba, Ibaraki 305-8565, Japan
matsuoka@ni.aist.go.jp

Abstract. In this paper, we provide direct encodings into Horn sequents of Multiplicative Linear Logic for two NP-complete problems, 3D MATCHING and PARTITION. Their correctness proofs are given by using a characterization of multiplicative proof nets.

1 Introduction

Around early 1990s, Max Kanovich introduced several Horn fragments of Linear Logic (see [1, 2]). In particular, the multiplicative Horn fragment (for short HMLL) is a rather restricted subsystem of Intuitionistic Multiplicative Linear Logic. In [2], HMLL is shown to be NP-complete with regard to provability through the encoding of the 3-PARTITION problem [3]. Moreover Krantz and Mogbil [4] gave another NP-completeness proof of HMLL through the encoding of the DIRECTED HAMILTONIAN CIRCUIT problem [3].

In this paper, we go forward further along this promising direction. We give the direct encoding of the 3D MATCHING problem as well as that of the PARTITION problem [3] into Horn sequents, which are those of HMLL, where a direct encoding means that it is obtained not through other encodings. It is well-known that an NP-complete problem can also solve any other NP-complete problem through polynomial time transformations. But direct encodings are important because they provide more efficient solvers for NP-complete problems than indirect encodings through polynomial time transformations. In fact problem sizes encoded directly tend to be significantly smaller than those encoded by polynomial time transformations from the viewpoint of a practical level, although both are related by polynomials.

NP-complete problems often appear in practical applications. To solve NP-complete problems is important: to obtain more optimized solutions of practical optimization problems means to reduce tangible resources more drastically for systems constructed based on these solutions. Recent progress of SAT solvers has enabled this at a practical level (for example, see [5]). We would like to address this topic from another logical point of view, i.e., from *Multiplicative Linear Logic* (for short MLL). In this approach we exploit *provability* of MLL instead of

satisfiability of classical logic. That is, in order to find a solution of an instance of an NP-complete problem, we must search for a proof for the MLL formula that is obtained from the instance by using an encoding for the NP-complete problem. Proof search over HMLL sequents can be regarded as multiset rewriting. In other words, a process solving NP-complete problems can be viewed as a sequence of actions over a commutative monoid. The approach is quite different from that of SAT solvers, since to solve problems encoded in SAT is to find solutions satisfying given static constraints. In author's opinion, it seems more natural to model an NP-complete problem as such a rewriting process, because a designer seems easier to specify a problem in such a modeling framework than to specify it in static constraints. In fact, the four direct encodings into Horn sequents mentioned above are surprisingly simple and easy to understand.

Moreover the software called *Proof Net Calculator* [6], which we have developed, includes an implementation of the 3D MATCHING problem as well as that of the DIRECTED HAMILTONIAN CIRCUIT problem, using the encodings mentioned above. Although we have not exploited *problem-specific* optimizations yet, Proof Net Calculator can solve instances of these problems which average persons (like the author) find difficult to solve, where we utilize *encoding-specific* optimizations using *ID-links dependency relations*. The technical details will be given elsewhere.

Proof Net Calculator is a versatile software for manipulating MLL proof nets. The way that MLL proof nets are represented in Proof Net Calculator is based on an idea used in [7, 8]. It also includes a normalization procedure based on Geometry of Interaction [9], a compiler for the linear lambda calculus [10] into MLL proof nets, and a translator of linear lambda terms into intuitionistic proof nets. In addition it also has a simple GUI tool, which displays an MLL proof net on a PC screen and outputs an encapsulated postscript file for it. Its implementation has been done using the programming language Scala [11], which is an object-oriented functional programming language over the Java Virtual Machine.

2 Intuitionistic Multiplicative Linear Logic, Horn Sequents, and MLL

2.1 Intuitionistic Multiplicative Linear Logic and Horn Sequents

In this section we introduce the system of Intuitionistic Multiplicative Linear Logic (for short IMLL) and then Horn sequents in IMLL. Two NP-complete problems, 3D MATCHING and PARTITION are encoded as Horn sequents as well as 3PARTITION in [2] and DIRECTED HAMILTONIAN CIRCUITS in [4]. Our terminology and notation with regard to sequent calculi are standard, although we exclude the cut rule, since we only deal with cut-free systems in this paper. For more technical details, for example, see [12]. We do not use the formulation of the multiplicative Horn fragment of Linear Logic (for short HMLL) described in [2]. That system includes the cut rule in an essential way: the cut elimination theorem does not hold in it. All we need in this paper are cut-free systems.

We denote *atomic formulas* by p, q, r, \dots . Then we define *IMLL formulas*, which are denoted by A, B, C, \dots by the following grammar:

$$A ::= p \mid A \multimap B \mid A \otimes B$$

We denote *multisets of IMLL formulas* by $\Sigma, \Sigma_1, \Sigma_2, \dots$. An IMLL sequent is a pair (Σ, A) . We write an IMLL sequent (Σ, A) as $\Sigma \vdash A$. The inference rules of IMLL are as follows:

$$\begin{array}{l} \text{I} \quad \overline{A \vdash A} \\ \text{L} \multimap \frac{\Sigma_1 \vdash A \quad B, \Sigma_2 \vdash C}{\Sigma_1, A \multimap B, \Sigma_2 \vdash C} \quad \text{R} \multimap \frac{\Sigma, A \vdash B}{\Sigma \vdash A \multimap B} \\ \text{L} \otimes \frac{\Sigma, A, B \vdash C}{\Sigma, A \otimes B \vdash C} \quad \text{R} \otimes \frac{\Sigma_1 \vdash A \quad \Sigma_2 \vdash B}{\Sigma_1, \Sigma_2 \vdash A \otimes B} \end{array}$$

By a *simple formula* we mean a formula that consists of only atomic formulas and \otimes connective. We denote simple formulas by X, Y, Z, \dots . By a *Horn implication* we mean a formula that has the form $X \multimap Y$. A *Horn sequent* is an IMLL sequent that has the form $X, \Sigma \vdash Y$, where each formula in Σ is a Horn implication. We note that the Horn sequents are a rather restricted class of the IMLL sequents.

2.2 Multiplicative Linear Logic

Next we introduce the system of Multiplicative Linear Logic (for short MLL). We define *MLL formulas*, which are denoted by F, G, H, \dots , by the following grammar:

$$F ::= p \mid p^\perp \mid F \otimes G \mid F \wp G$$

The negation of F , which is denoted by F^\perp is defined as follows:

$$\begin{aligned} (p)^\perp &= p^\perp \\ (F \otimes G)^\perp &= G^\perp \wp F^\perp \\ (F \wp G)^\perp &= G^\perp \otimes F^\perp \end{aligned}$$

We denote *multisets of MLL formulas* by $\Lambda, \Lambda_1, \Lambda_2, \dots$. An MLL sequent is a multiset of MLL formulas Λ . We write an MLL sequent Λ as $\vdash \Lambda$. The inference rules of MLL are as follows:

$$\begin{array}{l} \text{ID} \quad \overline{\vdash F^\perp, F} \\ \otimes \quad \frac{\vdash \Lambda_1, F \quad \vdash \Lambda_2, G}{\vdash \Lambda_1, \Lambda_2, F \otimes G} \quad \wp \quad \frac{\vdash \Lambda, F, G}{\vdash \Lambda, F \wp G} \end{array}$$

We define a translation from IMLL sequents to MLL sequents by

$$A_1, \dots, A_n \vdash B \quad \mapsto \quad \vdash (A_1)^\perp, \dots, (A_n)^\perp, B^+$$

where $(-)^-$ and $(-)^+$ are defined inductively as follows:

$$\begin{aligned} (p)^- &= p^\perp & (p)^+ &= p \\ (A \multimap B)^- &= (B)^- \otimes (A)^+ & (A \multimap B)^+ &= (A)^- \wp (B)^+ \\ (A \otimes B)^- &= (B)^- \wp (A)^- & (A \otimes B)^+ &= (A)^+ \otimes (B)^+ \end{aligned}$$

Proposition 1. *A sequent $A_1, \dots, A_n \vdash B$ is provable in IMLL if and only if $\vdash (A_1)^-, \dots, (A_n)^-, B^+$ is provable in MLL.*

Proof. For example, see [13]. □

By Proposition 1 we can discuss Horn sequent encodings of NP-complete problems in the framework of MLL.

2.3 MLL Proof Nets

Next we introduce MLL proof nets. We use MLL proof nets in order to prove the correctness of our encodings into Horn sequents.

Figure 1 shows the *MLL links* we use. Each MLL link has a few MLL formulas. Such an MLL formula is a conclusion or a premise of the MLL link, which is specified as follows:

1. In an ID-link, each of F and F^\perp is called a conclusion of the link;
2. In an \otimes -link, each of F and G is called a premise of the link and $F \otimes G$ is called a conclusion of the link;
3. In an \wp -link, each of F and G is called a premise of the link and $F \wp G$ is called a conclusion of the link.

An MLL *proof structure* Θ is a set of MLL links that satisfies the following conditions:

1. For each link L in Θ , each conclusion of L can be a premise of at most one link other than L in Θ ;
2. For each link L in Θ , each premise of L must be a conclusion of exactly one link other than L in Θ .

An MLL *proof net* is an MLL proof structure that is constructed by the rules in Fig. 2. Note that each rule in Fig. 2 has the corresponding inference rule in the MLL sequent calculus. Any MLL proof structure is not necessarily an MLL proof net.

Next we introduce a characterization of MLL proof nets using the notion of DR-switchings. A DR-switching S for an MLL proof structure Θ is a function from the set of \wp -links in Θ to $\{0, 1\}$. The DR-graph $S(\Theta)$ for Θ and S is defined by the rules of Fig. 3. Then the following characterization holds.

Theorem 1 ([14]). *An MLL proof structure Θ is an MLL proof net if and only if for any DR-switching S for Θ , the DR-graph $S(\Theta)$ is acyclic and connected.*

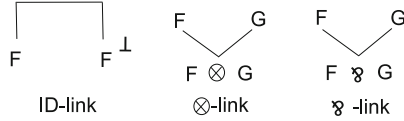


Fig. 1. MLL links

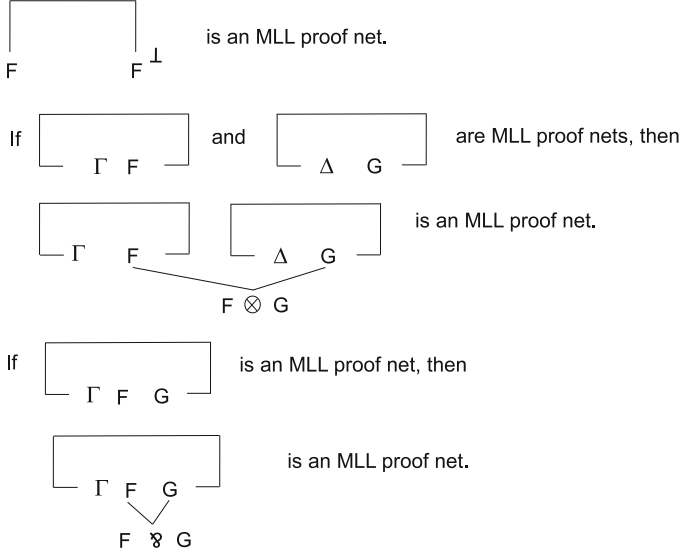


Fig. 2. Definition of MLL proof nets

Next we introduce a few notions for MLL proof search based on MLL proof nets. An MLL *proof forest* Θ_0 is a set of MLL links obtained from an MLL proof structure Θ by deleting all ID-links in Θ . An *ID-links set* π for an MLL proof forest Θ_0 is a set of ID-links such that $\Theta_0 \cup \pi$ is an MLL proof structure. Then we write $\Theta_0 \cup \pi$ as Θ_0^π . We say that an MLL proof forest Θ_0 has an MLL proof net if there is an ID-links set π for Θ_0 such that Θ_0^π has an MLL proof net. An MLL formula A is a conclusion of an MLL proof forest Θ_0 if there is a link L in Θ_0 such that A is a conclusion of L and there is no link L' in Θ_0 such that A is a premise of L' .

Proposition 2. *Let $A_1, \dots, A_n \vdash B$ be an IMLL sequent and $(A_1)^-, \dots, (A_n)^-, B^+$ be the conclusions of an MLL proof forest Θ_0 . Then $A_1, \dots, A_n \vdash B$ is provable in IMLL if and only if Θ_0 has an MLL proof net.*

Proof. By Proposition 1, $A_1, \dots, A_n \vdash B$ is provable in IMLL if and only if $\vdash (A_1)^-, \dots, (A_n)^-, B^+$ is provable in MLL. Moreover, if $\vdash (A_1)^-, \dots, (A_n)^-, B^+$ is provable in MLL then the MLL proof forest with the conclusions $(A_1)^-, \dots, (A_n)^-, B^+$ has an MLL proof net by the definition of MLL proof nets. The other direction is proved by Girard's sequentialization theorem [15]. \square

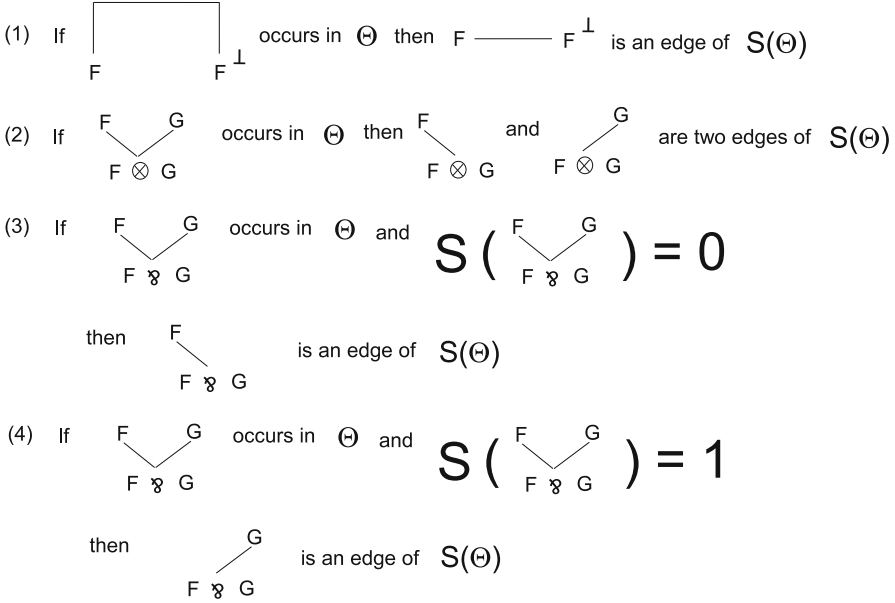


Fig. 3. Definition of DR graphs

3 The Encoding of 3D MATCHING

3.1 Preliminaries

Notation 1 Let S be a set. Let $L(k)$ be the set of all lists over S such that each list in $L(k)$ has the same length k . Then we define an equivalence relation Perm over $L(k)$ as follows: (ℓ_1, ℓ_2) is in Perm if for each $s \in S$, the number of occurrences of s in ℓ_1 is the same as that of ℓ_2 . Each equivalence class over Perm is a multiset. When ℓ is a list over S , let ℓ/Perm be the multiset that includes ℓ .

Definition 1 (3D MATCHING [16]). Let A, B, C be finite sets such that $|A| = |B| = |C| = n$. 3D MATCHING is the problem that when a given set $T \subseteq A \times B \times C$, decides whether or not there is a subset T_0 of T such that $|T_0| = n$ and

$$\begin{aligned} A &= \{a \in A \mid \exists b \in B. \exists c \in C. \langle a, b, c \rangle \in T_0\} \\ B &= \{b \in B \mid \exists a \in A. \exists c \in C. \langle a, b, c \rangle \in T_0\} \\ C &= \{c \in C \mid \exists a \in A. \exists b \in B. \langle a, b, c \rangle \in T_0\}. \end{aligned}$$

We suppose $|T| = n + m$, where $m \geq 0$. We assume that the elements of T are ordered and the list is

$$\langle a_{i_1}, b_{j_1}, c_{k_1} \rangle, \dots, \langle a_{i_n}, b_{j_n}, c_{k_n} \rangle, \langle a_{i_{n+1}}, b_{j_{n+1}}, c_{k_{n+1}} \rangle, \dots, \langle a_{i_{n+m}}, b_{j_{n+m}}, c_{k_{n+m}} \rangle.$$

Then we define three *multisets* A_T, B_T, C_T by

$$\begin{aligned} A_T &= \langle a_{i_1}, \dots, a_{i_n}, a_{i_{n+1}}, \dots, a_{i_{n+m}} \rangle / \text{Perm} \\ B_T &= \langle b_{j_1}, \dots, b_{j_n}, b_{j_{n+1}}, \dots, b_{j_{n+m}} \rangle / \text{Perm} \\ C_T &= \langle c_{k_1}, \dots, c_{k_n}, c_{k_{n+1}}, \dots, c_{k_{n+m}} \rangle / \text{Perm} \end{aligned}$$

Without loss of generality, we suppose the following conditions:

1. For each $a \in A$, there is ℓ ($1 \leq \ell \leq n + m$) such that $a = a_{i_\ell}$;
2. For each $b \in B$, there is ℓ ($1 \leq \ell \leq n + m$) such that $b = b_{j_\ell}$;
3. For each $c \in C$, there is ℓ ($1 \leq \ell \leq n + m$) such that $c = c_{k_\ell}$.

Otherwise, we can determine that this instance has no solution. Moreover we define three *multisets* $A_{\text{co}}, B_{\text{co}}, C_{\text{co}}$ by

$$\begin{aligned} A_{\text{co}} &= A_T - A_{\text{mul}} \\ B_{\text{co}} &= B_T - B_{\text{mul}} \\ C_{\text{co}} &= C_T - C_{\text{mul}} \end{aligned}$$

where $A_{\text{mul}}, B_{\text{mul}}, C_{\text{mul}}$ are the multisets that have the same elements as A, B, C respectively such that each element occurs exactly once in $A_{\text{mul}}, B_{\text{mul}}, C_{\text{mul}}$ respectively. So, $|A_{\text{mul}}| = |B_{\text{mul}}| = |C_{\text{mul}}| = n$. Then without loss of generality, we can describe as

$$\begin{aligned} A_{\text{mul}} &= \langle a_1, \dots, a_n \rangle / \text{Perm} & A_{\text{co}} &= \langle a_{i'_1}, \dots, a_{i'_m} \rangle / \text{Perm} \\ B_{\text{mul}} &= \langle b_1, \dots, b_n \rangle / \text{Perm} & B_{\text{co}} &= \langle b_{j'_1}, \dots, b_{j'_m} \rangle / \text{Perm} \\ C_{\text{mul}} &= \langle c_1, \dots, c_n \rangle / \text{Perm} & C_{\text{co}} &= \langle c_{k'_1}, \dots, c_{k'_m} \rangle / \text{Perm} \end{aligned}$$

3.2 The Encoding into a Horn Sequent

In this section we give our encoding of 3D MATCHING problem into a Horn sequent. We need a few auxiliary formulas. For each ℓ ($1 \leq \ell \leq n + m$), we define FT_ℓ by

$$FT_\ell = (b_{j_\ell} \otimes c_{k_\ell}) \multimap a_{i_\ell}$$

Moreover we define $FA_{\text{co}}, FB_{\text{co}}, FC_{\text{co}}$ by

$$\begin{aligned} FA_{\text{co}} &= a_{i'_1} \otimes \dots \otimes a_{i'_m} \\ FB_{\text{co}} &= b_{j'_1} \otimes \dots \otimes b_{j'_m} \\ FC_{\text{co}} &= c_{k'_1} \otimes \dots \otimes c_{k'_m} \end{aligned}$$

Finally we define F_1 by

$$F_1 = FA_{\text{co}} \multimap ((b_1 \otimes \dots \otimes b_n) \otimes (c_1 \otimes \dots \otimes c_n))$$

Then we define a sequent as

$$\Gamma_{\text{3DMATCHING}} = FB_{\text{co}} \otimes FC_{\text{co}}, F_1, FT_1, \dots, FT_n, FT_{n+1}, \dots, FT_{n+m} \vdash a_1 \otimes \dots \otimes a_n$$

It is obvious that $\Gamma_{\text{3DMATCHING}}$ is a Horn sequent and the encoding is a polynomial reduction.

Example 1. The following is an instance of 3D MATCHING:

$$\begin{aligned}
 A &= \{a_1, a_2, a_3\} \\
 B &= \{b_1, b_2, b_3\} \\
 C &= \{c_1, c_2, c_3\} \\
 T &= \{\langle a_1, b_1, c_2 \rangle, \langle a_1, b_2, c_3 \rangle, \langle a_2, b_2, c_1 \rangle, \langle a_2, b_3, c_1 \rangle, \langle a_3, b_1, c_2 \rangle\}
 \end{aligned}$$

The encoding described above gives the following Horn sequent from the instance:

$$\begin{aligned}
 \Gamma_{\text{3DMATCHING}} = & \\
 & (b_1 \otimes b_2) \otimes (c_1 \otimes c_2), \\
 & a_1 \otimes a_2 \multimap ((b_1 \otimes b_2 \otimes b_3) \otimes (c_1 \otimes c_2 \otimes c_3)), \\
 & b_1 \otimes c_2 \multimap a_1, \quad b_2 \otimes c_3 \multimap a_1, \quad b_2 \otimes c_1 \multimap a_2, \\
 & b_3 \otimes c_1 \multimap a_2, \quad b_1 \otimes c_2 \multimap a_3 \\
 & \vdash a_1 \otimes a_2 \otimes a_3
 \end{aligned}$$

3.3 The Correctness Proof

In order to prove the correctness of the encoding, we exploit the characterization of MLL proof nets (Theorem 1). We construct an MLL proof forest, which corresponds to the Horn sequent $\Gamma_{\text{3DMATCHING}}$.

For each ℓ ($1 \leq \ell \leq n + m$), the forest Θ_ℓ corresponding to FT_ℓ is shown in Fig. 4. The forest Θ_I corresponding to F_I is shown in Fig. 5. Then the forest Θ_F corresponding to the rest in $\Gamma_{\text{3DMATCHING}}$ is shown in Fig. 6. Then we define an MLL proof forest Θ_0 as

$$\Theta_0 = \Theta_I \cup \bigcup_{1 \leq \ell \leq n+m} \Theta_\ell \cup \Theta_F$$

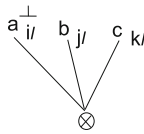


Fig. 4. Triple device Θ_ℓ

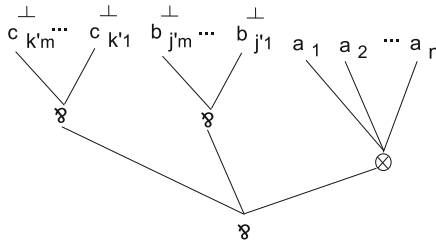


Fig. 5. I-device Θ_I

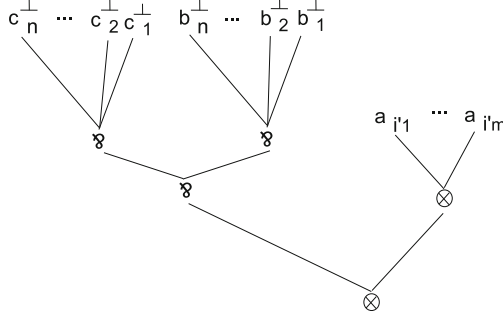


Fig. 6. F-device Θ_F

Then if we note that $\vdash \Delta, A \wp B$ is provable in MLL if and only if $\vdash \Delta, A, B$ is provable in MLL, then we can easily see that $\Gamma_{3\text{DMATCHING}}$ is provable in IMLL if and only if Θ_0 has an MLL proof net by Proposition 2.

Theorem 2. *An instance of 3D MATCHING has a solution if and only if there is an ID-links set π for the corresponding MLL proof forest Θ_0 such that Θ_0^π is an MLL proof net.*

Proof. We assume that we have a solution $T_0 \subseteq A \times B \times C$. Then without loss of generality we can write as

$$T_0 = \{\langle a_{i_1}, b_{j_1}, c_{k_1} \rangle, \dots, \langle a_{i_n}, b_{j_n}, c_{k_n} \rangle\}.$$

Moreover we have

$$\begin{aligned} A &= \{a_{i_1}, \dots, a_{i_n}\} \\ B &= \{b_{j_1}, \dots, b_{j_n}\} \\ C &= \{c_{k_1}, \dots, c_{k_n}\}. \end{aligned}$$

Then for each ℓ ($1 \leq \ell \leq n$), there is exactly one triple device Θ_{q_ℓ} corresponding to $\langle a_{i_\ell}, b_{j_\ell}, c_{k_\ell} \rangle$ in T_0 where $1 \leq q_\ell \leq n + m$. Let

$$\mathcal{T}_0 = \{\Theta_{q_1}, \dots, \Theta_{q_n}\}$$

Moreover we can write the set of the triple devices that do not appear in \mathcal{T}_0 as

$$\mathcal{T}_1 = \{\Theta_{q_{n+1}}, \dots, \Theta_{q_{n+m}}\}$$

We construct an ID-links set π for Θ_0 as follows:

- (1) For each ℓ ($1 \leq \ell \leq n$) π includes the ID-link that connects the literal $a_{i_\ell}^\perp$ in the triple device Θ_{q_ℓ} to a_{i_ℓ} in the I-device Θ_I .
- (2) For each ℓ ($1 \leq \ell \leq n$) π includes the ID-link that connects the literal b_{j_ℓ} in the triple device Θ_{q_ℓ} to $b_{j_\ell}^\perp$ in the F-device Θ_F .

- (3) For each ℓ ($1 \leq \ell \leq n$) π includes the ID-link in π that connects the literal c_{k_ℓ} in the triple device Θ_{q_ℓ} to $c_{k_\ell}^\perp$ in the F-device Θ_F .
- (4) For each ℓ ($n+1 \leq \ell \leq n+m$) π includes an ID-link that connects the literal $a_{i_\ell}^\perp$ in the triple device Θ_{q_ℓ} to a literal a_{i_ℓ} in the F-device Θ_F .
- (5) For each ℓ ($n+1 \leq \ell \leq n+m$) π includes an ID-link that connects the literal b_{j_ℓ} in the triple device Θ_{q_ℓ} to a literal $b_{j_\ell}^\perp$ in the I-device Θ_I .
- (6) For each ℓ ($n+1 \leq \ell \leq n+m$) π includes an ID-link that connects the literal c_{k_ℓ} in the triple device Θ_{q_ℓ} to a literal $c_{k_\ell}^\perp$ in the I-device Θ_I .

Then we can easily see that Θ_0^π is an MLL proof net.

Conversely we assume that we do not have any solution $T_0 \subseteq A \times B \times C$.

In this case we can not find the ID-links set π for Θ_0 described above. So any ID-links set π for Θ_0 must have the following property: There is some ℓ ($1 \leq \ell \leq n+m$) such that

- (1) the literal $a_{i_\ell}^\perp$ in Θ_ℓ connects to a_{i_ℓ} of the F-device Θ_F in π , and
- (2) the literal b_{j_ℓ} in Θ_ℓ connects to $b_{j_\ell}^\perp$ of the F-device Θ_F in π or
- (2') the literal c_{k_ℓ} in Θ_ℓ connects to $c_{k_\ell}^\perp$ of the F-device Θ_F in π .

Then we can find a DR-switching S for Θ_0^π such that the DR-graph $S(\Theta_0^\pi)$ has a cycle that passes through $a_{i_\ell}^\perp, a_{i_\ell}$ and $b_{j_\ell}^\perp, b_{j_\ell}$, or $a_{i_\ell}^\perp, a_{i_\ell}$ and $c_{k_\ell}^\perp, c_{k_\ell}$. \square

Remark 1. In fact our proof of Theorem 2 gives a stronger statement: if we find an MLL proof net Θ_0^π for Θ_0 then we can separate a matching T_0 from T using the ID-link set π .

Corollary 1. *An instance of 3D MATCHING has a solution if and only if the sequent $\Gamma_{\text{3DMATCHING}}$ for the instance is provable in MLL.*

We note that this result can be easily extended to the n-D MATCHING problem for any n ($n \geq 2$).

4 The Encoding of PARTITION

4.1 Preliminaries

Definition 2 (PARTITION [16]). *Let A be a finite set and s be a function from A to \mathbb{Z}^+ . PARTITION is the problem that decide whether or not there is a subset $A' \subseteq A$ such that*

$$\sum_{s \in A'} s(a) = \sum_{a \in A - A'} s(a).$$

The problem is different from 3-PARTITION used in [2] and an NP-complete problem [3]. In particular our encoding below cannot be derived from the 3-PARTITION encoding in [2] directly.

We assume that the elements of A are ordered and the list is

$$a_1, a_2, \dots, a_k$$

Let

$$t = \sum_{1 \leq i \leq k} s(a_i).$$

Note that for any subset $A' \subseteq A$,

$$t = \sum_{s \in A'} s(a) + \sum_{a \in A - A'} s(a).$$

If A' is a solution, then the following equation must hold:

$$t = 2 \sum_{s \in A'} s(a).$$

Then t must be even. So without loss of generality, we can assume t is even.

4.2 The Encoding into a Horn Sequent

In this section we give our encoding of the PARTITION problem into a Horn sequent. We need a few auxiliary formulas.

For each i ($1 \leq i \leq k$), we define $F_{\text{one}i}$ and $F_{\text{ano}i}$ as

$$F_{\text{one}i} = a_i \multimap \overbrace{b \otimes b \otimes \cdots \otimes b}^{s(a_i)}$$

$$F_{\text{ano}i} = a_i \multimap \overbrace{c \otimes c \otimes \cdots \otimes c}^{s(a_i)}$$

Let F_{weight} be

$$F_{\text{weight}} = \left(\overbrace{b \otimes b \otimes \cdots \otimes b}^{t/2} \right) \otimes \left(\overbrace{c \otimes c \otimes \cdots \otimes c}^{t/2} \right)$$

We define $F_{1\text{st}}$ and $F_{2\text{nd}}$ as

$$F_{1\text{st}} = F_{\text{weight}} \multimap a_1 \otimes a_2 \otimes \cdots \otimes a_k$$

$$F_{2\text{nd}} = F_{\text{weight}} \multimap e$$

Then we define a sequent as

$$\Gamma_{\text{PARTITION}} = a_1 \otimes a_2 \otimes \cdots \otimes a_k, F_{\text{one}1}, \dots, F_{\text{one}k}, F_{\text{ano}1}, \dots, F_{\text{ano}k}, F_{1\text{st}}, F_{2\text{nd}} \vdash e$$

It is obvious that $\Gamma_{\text{PARTITION}}$ is a Horn sequent and the encoding is a polynomial reduction.

We give an informal meaning for these formulas as follows:

- The formula $a_1 \otimes a_2 \otimes \cdots \otimes a_k$ gives the multiset of all items.
- The formula $F_{\text{one}i}$ gives the weight for the item a_i .

- The formula F_{anoi} also gives the weight for the item a_i .
- The role of F_{1st} is a balance. If we are able to partition the set A of the items into two disjoint sets A' and $A - A'$ such that the sum of weights of A' is equal to that $A - A'$, then again we get the multiset of all items $a_1 \otimes a_2 \otimes \cdots \otimes a_k$.
- The role of F_{2nd} is also a balance. If we succeed in the partition mentioned above, then we get the final formula e .

Example 2. The following is an instance of PARTITION:

$$A = \{a_1, a_2, a_3, a_4\}, \quad s = \{a_1 \mapsto 2, a_2 \mapsto 3, a_3 \mapsto 2, a_4 \mapsto 1\}$$

The encoding described above gives the following Horn sequent from the instance:

$$\begin{aligned} \Gamma_{\text{PARTITION}} = & \\ & a_1 \otimes a_2 \otimes a_3 \otimes a_4, \\ & a_1 \multimap b \otimes b, \quad a_2 \multimap b \otimes b \otimes b, \quad a_3 \multimap b \otimes b, \quad a_4 \multimap b, \\ & a_1 \multimap c \otimes c, \quad a_2 \multimap c \otimes c \otimes c, \quad a_3 \multimap c \otimes c, \quad a_4 \multimap c, \\ & (b \otimes b \otimes b \otimes b) \otimes (c \otimes c \otimes c \otimes c) \multimap a_1 \otimes a_2 \otimes a_3 \otimes a_4, \\ & (b \otimes b \otimes b \otimes b) \otimes (c \otimes c \otimes c \otimes c) \multimap e \qquad \qquad \qquad \vdash e \end{aligned}$$

4.3 The Correctness Proof

In order to prove the correctness of the encoding, we exploit the characterization of MLL proof nets (Theorem 1). We construct an MLL proof forest, which corresponds to the Horn sequent $\Gamma_{\text{PARTITION}}$. Let the MLL forest shown in Fig. 7 be Θ_I . For each i ($1 \leq i \leq k$), let the MLL forest shown in Fig. 8 be $\Theta_{\text{one}i}$. For each i ($1 \leq i \leq k$), let the MLL forest shown in Fig. 9 be $\Theta_{\text{anoi}i}$. Let the MLL forest shown in Fig. 10 be Θ_{1st} . Let the MLL forest shown in Fig. 11 be Θ_{2nd} . Finally let the MLL forest shown in Fig. 12 be Θ_F . Then we define an MLL proof forest Θ_0 as

$$\Theta_0 = \Theta_I \cup \bigcup_{1 \leq i \leq k} \Theta_{\text{one}i} \cup \bigcup_{1 \leq i \leq k} \Theta_{\text{anoi}i} \cup \Theta_{\text{1st}} \cup \Theta_{\text{2nd}} \cup \Theta_F$$

Then by Proposition 2, $\Gamma_{\text{PARTITION}}$ is provable in IMLL if and only if Θ_0 has an MLL proof net.

Theorem 3. *An instance of PARTITION has a solution if and only if there is an ID-links set π for the corresponding MLL proof forest Θ_0 such that Θ_0^π is an MLL proof net.*

Proof. We assume that we have a solution $A' \subseteq A$. Then without loss of generality we can write as

$$\begin{aligned} A' &= \{a_1, \dots, a_{k_0}\} \\ A - A' &= \{a_{k_0+1}, \dots, a_k\} \end{aligned}$$

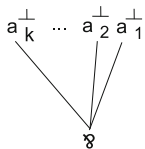


Fig. 7. I-device Θ_I

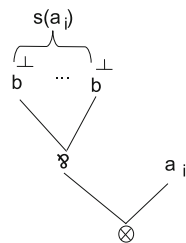


Fig. 8. One side device Θ_{onei}

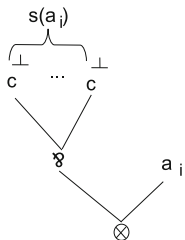


Fig. 9. Another side device Θ_{anoi}

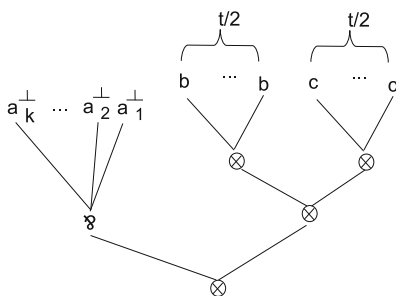


Fig. 10. The first matching device Θ_{1st}

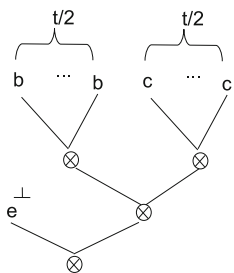


Fig. 11. The second matching device Θ_{2nd}

e

Fig. 12. F-device Θ_F

Since A' is a solution, as mentioned above, we can write as

$$t/2 = \sum_{s \in A'} s(a) = \sum_{s \in A - A'} s(a).$$

Then we construct an ID-links set π for Θ_0 as follows:

- (1) For each i ($1 \leq i \leq k_0$), a_i^\perp in Θ_I is connected to a_i in $\Theta_{\text{one}i}$ and each of b^\perp in $\Theta_{\text{one}i}$ is connected to b in $\Theta_{1\text{st}}$.
- (2) For each i ($1 \leq i \leq k_0$), a_i^\perp in $\Theta_{1\text{st}i}$ is connected to a_i in Θ_{anoi} and each of c^\perp in Θ_{anoi} is connected to c in $\Theta_{2\text{nd}}$.
- (3) For each i ($k_0 + 1 \leq i \leq k$), a_i^\perp in Θ_I is connected to a_i in Θ_{anoi} and each of c^\perp in Θ_{anoi} is connected to c in $\Theta_{1\text{st}}$.
- (4) For each i ($k_0 + 1 \leq i \leq k$), a_i^\perp in $\Theta_{1\text{st}}$ is connected to a_i in $\Theta_{\text{one}i}$ and each of b^\perp in $\Theta_{\text{one}i}$ is connected to b in $\Theta_{2\text{nd}}$.
- (5) The literal e^\perp in $\Theta_{2\text{nd}}$ is connected to e in Θ_F .

It is obvious that Θ_0^π is an MLL proof net.

Conversely, we assume that we do not have any solution $A' \subseteq A$. This means that for any subset $A' \subseteq A$,

$$t/2 \neq \sum_{s \in A'} s(a) \neq \sum_{s \in A - A'} s(a) \neq t/2.$$

Then any ID-links set π for Θ_0 must have the following property:

There is some i ($1 \leq i \leq k$) such that

- (1) the literal a_i^\perp in $\Theta_{1\text{st}}$ is connected to the literal a_i in $\Theta_{\text{one}i}$;
 - (2) a literal b in $\Theta_{1\text{st}}$ is connected to a literal b^\perp in $\Theta_{\text{one}i}$,
- or
- (1') the literal a_i^\perp in $\Theta_{1\text{st}}$ is connected to the literal a_i in Θ_{anoi} ;
 - (2') a literal c in $\Theta_{1\text{st}}$ is connected to a literal c^\perp in Θ_{anoi} .

Then there is a DR-switching S for Θ_0^π such that $S(\Theta_0^\pi)$ has a cycle including all literals mentioned in (1) and (2), or (1') and (2'). \square

Remark 2. In fact our proof of Theorem 3 gives a stronger statement: if we find an MLL proof net Θ_0^π for Θ_0 then we can separate a partition A' from A using the ID-link set π .

Corollary 2. *An instance of PARTITION has a solution if and only if the sequent $\Gamma_{\text{PARTITION}}$ for the instance is provable in MLL.*

Corollary 2 can be easily generalized to that for the following problem.

Definition 3 (GENERAL PARTITION). *Let A be a finite set and s be a function from A to \mathbb{Z}^+ . Let*

$$t = \sum_{s \in A} s(a)$$

and n_1, n_2, \dots, n_ℓ be a list of positive integers such that

$$t = n_1 + n_2, \dots + n_\ell.$$

GENERAL PARTITION is the problem that decide whether or not there is a list of pairwise disjoint subsets A_1, A_2, \dots, A_ℓ of A such that

$$A = \bigcup_{1 \leq \ell' \leq \ell} A_{\ell'}$$

and

$$n_1 = \sum_{s \in A_1} s(a), n_2 = \sum_{s \in A_2} s(a), \dots, n_\ell = \sum_{s \in A_\ell} s(a).$$

The idea of the encoding is that we prepare ℓ side devices instead of one side and another side devices, increase the number of the matching devices from 2 to ℓ , and use a cycle permutation of ℓ -cycle on ℓ letters.

The **GENERAL PARTITION** problem can be regarded as a special version of the **BIN PACKING** problem [3]. It has several practical applications: for example, imagine that you want to make a backup copy of a huge number of data files on your PC into several different relatively small storage devices like USB flash memory devices. Then we have got an instance of the problem, where $s : A \rightarrow \mathbb{Z}^+$ is a function from file identifiers to file sizes.

5 Concluding Remarks

In this paper we showed that Horn sequents of Linear Logic are extremely useful for formalizing combinatorial NP-completeness problems. This suggests that more complicated practical combinatorial NP-complete problems can be directly encoded into these sequents. In fact, we have already discovered direct encodings of several other NP-complete problems into MLL, including **CIRCUIT SATISFACTION** (with constant fan-in), **3SAT**, **UNDIRECTED HAMILTONIAN CIRCUIT**, and **GRAPH 3-COLORABILITY**.

Although Kanovich's Horn sequents of Linear Logic has not received much attention so far, we believe that the research direction is promising.

References

1. Kanovich, M.I.: Horn programming in linear logic is NP-complete. In: Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science, pp. 200–210 (1992)
2. Kanovich, M.I.: The complexity of Horn fragments of linear logic. *Ann. Pure Appl. Logic* **69**, 195–241 (1994)
3. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York (1979)
4. Krantz, T., Mogbil, V.: Encoding Hamiltonian circuits into multiplicative linear logic. *Theoret. Comput. Sci.* **266**, 987–996 (2001)

5. Malik, S., Zhang, L.: Boolean satisfiability: from theoretical hardness to practical success. *Commun. ACM* **52**, 76–82 (2009)
6. Matsuoka, S.: Proof Net Calculator (2017). <https://staff.aist.go.jp/s-matsuoka/PNCalculator/index.html>
7. Matsuoka, S.: Weak typed Böhm theorem on IMLL. *Ann. Pure Appl. Logic* **145**(1), 37–90 (2007)
8. Matsuoka, S.: A coding theoretic study of MLL proof nets. *Math. Struct. Comput. Sci.* **22**(3), 409–449 (2012)
9. Girard, J.Y.: Multiplicatives. In: *Logic and Computer Science: New Trends and Applications*, pp. 11–34 (1988)
10. Matsuoka, S.: Strong typed Böhm theorem and functional completeness on the linear lambda calculus. In: *Proceedings of 6th Workshop on Mathematically Structured Functional Programming, MSFP 2016*, pp. 1–22 (2016)
11. Odersky, M., Spoon, L., Venners, B.: *Programming in Scala*, 3rd edn. Artima Inc., Walnut Creek (2016)
12. Girard, J.Y., Lafont, Y., Taylor, P.: *Proofs and Types*. Cambridge University Press, Cambridge (1989)
13. Murawski, A.M., Ong, C.H.L.: Fast verification of MLL proof nets via IMLL. *ACM Trans. Comput. Logic* **7**, 473–498 (2006)
14. Danos, V., Regnier, R.: The structure of multiplicatives. *Arch. Math. Logic* **28**, 181–203 (1989)
15. Girard, J.Y.: Linear logic. *Theoret. Comput. Sci.* **50**, 1–102 (1987)
16. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W., Bohlinger, J.D. (eds.) *Complexity of Computer Computations: Proceedings of a Symposium on the Complexity of Computer Computations*, pp. 85–103. Springer, Boston (1972)



λ to SKI, Semantically Declarative Pearl

Oleg Kiselyov^(✉)

Tohoku University, Sendai, Japan
oleg@okmij.org

Abstract. We present a technique for compiling lambda-calculus expressions into SKI combinators. Unlike the well-known bracket abstraction based on (syntactic) term re-writing, our algorithm relies on a specially chosen, *compositional* semantic model of generally open lambda terms. The meaning of a closed lambda term is the corresponding SKI combination. For simply-typed as well as untyped terms, the meaning derivation mirrors the typing derivation. One may also view the algorithm as an algebra, or a non-standard evaluator for lambda-terms (i.e., denotational semantics).

The algorithm is implemented as a tagless-final compiler for (uni)typed lambda-calculus embedded as a DSL into OCaml. Its type preservation is clear even to OCaml. The correctness of both the algorithm and of its implementation becomes clear.

Our algorithm is easily amenable to optimizations. In particular, its output and the running time can both be made linear in the size (i.e., the number of all constructors) of the input De Bruijn-indexed term.

1 Introduction

Since Curry [2] definitely and constructively demonstrated that any lambda-expression can be transformed into SKI-combinators, there seems to have been no need to revisit this issue. And yet it has continued to attract attention: of mathematicians, investigating connections of lambda-terms and graphs, and of theoretical computer scientists, studying the complexity of this process [4].

In a surprising turn, the translation from lambda-terms to SKI combinators, previously regarded as purely academic, proved very practical. David Turner used the translation as the compilation technique for his functional language SASL [12,13], and later Miranda. The familiar presentation of the translation, or compilation – called “the bracket abstraction” and originally due to Schoenfinkel [8] – was made popular by Turner, who polished and optimized it, and made it practical. The entire chapter of Peyton Jones’ book on implementing functional languages [7, Chap. 16] is devoted to the SKI translation, which is “appealing because it gives rise to an extremely simple reduction machine”. The book mentions two physical machines designed around SKI reductions: Cambridge SKIM machine [11] and Burroughs NORMA.

Above all, the bracket abstraction is a pearl. We can't help to peek at its shine right away, ahead of the formal presentation in the background Sect. 2. The translation turns a lambda-term to applications of the S, K, I combinators, whose reductions are shown in the left column of Fig. 1.

	Reductions	Compilation Rules
I	$Ix \rightsquigarrow x$	$\lambda x. x \mapsto I$
K	$Kyx \rightsquigarrow y$	$\lambda x. e \mapsto Ke \dagger$
S	$Sfgx \rightsquigarrow fx(gx)$	$\lambda x. e_1 e_2 \mapsto S (\lambda x. e_1) (\lambda x. e_2)$

Fig. 1. SKI reductions and compilation rules. \dagger In the K compilation rule, e is a combinator or variable other than x .

The translation is re-writing with three simple rules of the right column of Fig. 1. The I and K rules are the re-statements of the corresponding reductions; the S rule becomes obvious if we notice that a term e with a possibly free variable x is equal to $(\lambda x. e)x$. Taking as the running example $\lambda x. \lambda y. y x$, we can only use the S-rule, on the inner lambda-abstraction, obtaining $\lambda x. S (\lambda y. y) (\lambda y. x)$, to which K and I rules apply, giving $\lambda x. (SI) (Kx)$. Using the S rule three more times leads eventually to $S(S(KS)(KI))(S(KK)I)$. The result is bigger than the original term: we tackle the size explosion in Sect. 6.

We present another pearl of the translation from lambda-terms to SKI combinators and show off its facets. It comes from a very different oyster. Our translation is *not* based on (syntactic, in its essence) re-writing. Rather, we define a semantic model of (generally open) lambda-terms in terms of combinators, along with the way to compositionally compute the meaning of a term in that semantics from the meanings of its immediate children. The meaning of a closed lambda-term is designed to be the corresponding SKI term. Our translation stands out in avoiding operations like checking variable equality or free occurrences. Whereas the bracket abstraction cannot do anything meaningful with the mere x or $x y$ subterms, ours can. As a source we use lambda-terms with De Bruijn indices; it turns out the indices supply just enough information about the environment to figure out the meaning of a single variable or a combination of variables.

All in all, the semantic presentation of the SKI compilation avoids the ‘nominal trench’ of lambda-calculus; it is easier to see correct, easier to generalize and optimize. The semantic pearl shines brighter.

The Highlights

- Section 3 develops the semantic-based translation intuitively and formally. We start with the simply-typed calculus to see the correspondence of type and meaning derivations. Already the simplest polish naturally reveals optimizations.

- The semantic translation is straightforward to realize in tagless-final style: Sect. 4. The OCaml implementation highlights the algebra of the translation, naturally prompting further optimizations.
- Section 5 extends the calculus with integers, conditional and general recursion. The translation becomes practical.
- Section 6 presents the linear space and time translation, for the general untyped calculus (which applies to the typed calculus as well). It goes beyond the Schoenfinkel, Curry and Turner bracket abstraction and their further optimizations such as director strings; Sect. 7 discusses how much beyond and in which direction. Our translation is hence the viable alternative to supercombinators.

We start with the background, in the next section. The complete OCaml code is available at <http://okmij.org/ftp/tagless-final/skconv.ml>.

2 Lambda- and SKI-calculi and the Bracket Abstraction

This background section recapitulates lambda- and combinator calculi and the classical bracket abstraction. Mainly, it introduces the notation for the rest of the paper.

Figure 2 presents the syntax of the calculi and the notational conventions, heavily used throughout. We write e for expressions in lambda-calculus with names and e^0 for expressions in the calculus with De Bruijn indices – although we often write just e if the context disambiguates. Lower-case f, g, x, y, u, v (possibly adorned with subscripts or superscripts) are always variables. We consider both untyped calculi and simply-typed calculi. In the latter case, types (denoted by $\alpha, \beta, \gamma, \sigma, \tau$ metavariables) are base and arrow types; there are no type variables. Γ denotes a possibly empty sequence of types, whereas Γ^+ stands for a nonempty sequence. We write τ, Γ and Γ, τ for prepending, resp. appending τ to the sequence Γ , and Γ_1, Γ_2 for sequence concatenation. The S, K, I and other combinators are, by convention, upper-case letters; the metavariable d stands for an arbitrary combinator expression.

The meaning of combinators is defined by their reduction rules, collected in Fig. 3. It presents not only S, K, and I but also B and C combinators, which we

Variables	f, g, x, y, u, v
Base Types	ι
Types	$\alpha, \beta, \gamma, \sigma, \tau ::= \iota \mid \tau \rightarrow \tau$
Type Environment	$\Gamma ::= \tau, \dots$
Expressions	$e ::= x \mid \lambda x. e \mid e e$
De Bruijn Expressions	$e^0 ::= z \mid s e^0 \mid \lambda e^0 \mid e^0 e^0$
Combinator Expressions	$d ::= d d \mid S \mid K \mid I \mid B \mid C$

Fig. 2. Syntax of languages

$$\begin{array}{ll}
I x & \rightsquigarrow x \\
K y x & \rightsquigarrow y \\
S f g x & \rightsquigarrow fx(gx)
\end{array}
\qquad
\begin{array}{l}
B f g x \rightsquigarrow f(gx) \\
C f g x \rightsquigarrow fxg
\end{array}$$

Fig. 3. Combinators and their reduction rules

shall use later. They are particular cases of S; B is the functional composition. All combinators can be expressed in terms of just S and K.

Figure 4, borrowed from [7, Fig. 16.2] with the adjusted notation, presents the basic bracket abstraction algorithm: the formalization of the procedure intuitively described in Sect. 1. The main translation $\mathcal{C}[e]$ uses the auxiliary $\mathcal{A}_x[e']$ to “abstract” x from a lambda-free expression e' and produce the corresponding combinator expression. As shown in Sect. 1, $\mathcal{C}[\lambda x. \lambda y. y x]$ gives $S(S(KS)(KI))(S(KK)I)$.

$$\begin{array}{ll}
\mathcal{C}[e]: \text{ compile } e \text{ to combinators} & \mathcal{A}_x[e]: \text{ abstract } x \text{ from } e \\
\mathcal{C}[e_1 e_2] \mapsto \mathcal{C}[e_1] \mathcal{C}[e_2] & \mathcal{A}_x[e_1 e_2] \mapsto S(\mathcal{A}_x[e_1]) (\mathcal{A}_x[e_2]) \\
\mathcal{C}[\lambda x. e] \mapsto \mathcal{A}_x[\mathcal{C}[e]] & \mathcal{A}_x[x] \mapsto I \\
\mathcal{C}[c] \mapsto c & \mathcal{A}_x[c] \mapsto K c \quad \text{where } c \text{ is not } x
\end{array}$$

Fig. 4. Basic Bracket Abstraction. We write c for a variable, constant or a combinator expression. $\mathcal{A}_x[e]$ applies to e with no inner lambdas.

As another example, $\lambda y. (\lambda x. x x)(\lambda x. x x)$ is translated (in a less naive way) to $K((SII)(SII))$. The lambda-term is not strongly normalizing (i.e., has a divergent reduction sequence) and the same holds for the translated SKI term. With no reductions under lambda (as in call-by-name or call-by-value), the lambda-term is in normal form. In SKI this corresponds to the head reduction strategy: $d d_1 \dots d_n$, where d is a combinator, is irreducible if no reduction rule applies to d . In the following, only head reductions are considered.

3 Semantic Translation

This section formally presents our translation, first intuitively and then formally. We argue about its correctness in Sect. 4 and improving memory and run-time performance in Sect. 6. The translation is formulated as a non-standard denotation of lambda-terms and amounts to a constructive proof of the combinatorial completeness of lambda-calculus.

Although our translation applies both to typed and untyped calculi, the intuitions are easier to see with types, such as those in Fig. 5. (We explicitly consider the untyped case in Sect. 6.)

This is the very standard type system for simply-typed lambda-calculus, conventionally presented as the inference rules for the judgment $\Gamma \vdash e : \tau$ that

$$\begin{array}{c}
 \frac{}{\tau \vdash z : \tau} \text{Var} \quad \frac{\Gamma \vdash e : \tau}{\sigma, \Gamma \vdash e : \tau} \text{WL} \quad \frac{\Gamma^+ \vdash e : \tau}{\Gamma^+, \sigma \vdash s e : \tau} \text{WR} \\
 \\
 \frac{\Gamma, \sigma \vdash e : \tau}{\Gamma \vdash \lambda e : \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau} \text{App}
 \end{array}$$

Fig. 5. The simple type system of the De Bruijn lambda-calculus

a term e has the type τ in the environment Γ . The latter is a *sequence* of types. Figure 5 is more explicit than usual about structural rules, distinguishing the right weakening¹ (WR), which is syntactically marked as $s e$, from the left weakening (WL), which is unmarked. The ‘unhinged’ nature of (WL) prompts us to move to a different type system, in which every rule is connected to some syntactic feature and derivations are syntax-directed. The left-hand-side of Fig. 6 describes such a system – to be called ‘leftless’, in contrast with the ‘lefty’ system of Fig. 5, from which it was derived by working the (WL) rule into the others. The new type system is equivalent to the old, as the following proposition shows.

Definition 1. *A type judgment $\Gamma \vdash e : \tau$ (and its derivation) are called left-strong if (i) Γ is empty, or (ii) Γ is σ, Γ' and $\Gamma' \vdash e : \tau$ is not derivable.*

Proposition 1. *Any left-strong judgment (derivation) in the lefty system can be derived (converted to) the leftless system, and vice versa.*

In the forward direction, the proof is by the straightforward induction on the type derivation. The reverse direction is trivial.

The reason to split hairs about the weakening and to introduce the messier leftless system is to make clearer the correspondence between type and semantic derivations.

We now introduce our denotational semantics. Generally, denotational semantics assigns each syntactic object an element of some semantic domain, which serves as the ‘meaning’ for the object. The assignment must be compositional: the meaning of an object should depend only on the meanings of its immediate subcomponents. In Church-style calculi, only well-typed terms ‘make sense’. Therefore, the meaning is assigned to type derivations, represented by the judgement in their conclusion: in symbols, $\mathcal{E}[\vdash e : \tau] \in \mathcal{V}[\tau]$ where $\mathcal{V}[-]$ stands for the semantic domain, also type-indexed. To give the meaning to an open term, however, we need to know what its free variables mean. The common approach is to take the term denotation to be a function of an ‘environment’, which maps each free variable to its meaning. When variables are represented by De Bruijn indices, the environment may be realized as a tuple: $\mathcal{E}[\tau_n, \dots, \tau_1 \vdash e : \tau] \in \mathcal{V}[\tau_n \times \dots \times \tau_1 \rightarrow \tau]$.

¹ By ‘weakening’ we mean a (structural) inference rule stating that adding more premises to hypotheses of a valid logical deduction preserves the validity.

$\Gamma \vdash e : \tau$	$\mathcal{E}[\Gamma \vdash e : \tau]$
$\frac{}{\tau \vdash z : \tau} \text{Var}$	$\frac{}{\tau \models I} \text{EVar}$
$\frac{\Gamma^+ \vdash e : \tau}{\Gamma^+, \sigma \vdash s e : \tau} W$	$\frac{\Gamma^+ \models d}{\Gamma^+, \sigma \models (\models K) \Pi (\Gamma^+ \models d)} EW$
$\frac{\vdash e : \tau}{\vdash \lambda e : \sigma \rightarrow \tau} \text{Abs}_0$	$\frac{\models d}{\models K d} \text{EAbs}_0$
$\frac{\Gamma, \sigma \vdash e : \tau}{\Gamma \vdash \lambda e : \sigma \rightarrow \tau} \text{Abs}$	$\frac{\Gamma, \sigma \models d}{\Gamma \models d} \text{EAbs}$
$\frac{\Gamma_1 \vdash e_1 : \sigma \rightarrow \tau \quad \Gamma_2 \vdash e_2 : \sigma}{\Gamma_1 \sqcup \Gamma_2 \vdash e_1 e_2 : \tau} \text{App}$	$\frac{\Gamma_1 \models d_1 \quad \Gamma_2 \models d_2}{\Gamma_1 \sqcup \Gamma_2 \models (\Gamma_1 \models d_1) \Pi (\Gamma_2 \models d_2)} \text{EApp}$
where	$\Gamma_1 \sqcup \Gamma_2 = \Gamma_1$ if $\Gamma_1 = \Gamma_3, \Gamma_2$ for some Γ_3 $= \Gamma_2$ if $\Gamma_2 = \Gamma_3, \Gamma_1$

Fig. 6. The ‘leftless’ type system and the corresponding denotational semantics: the rules for deriving $\Gamma \vdash e : \tau$ (left column) and $\mathcal{E}[\Gamma \vdash e : \tau]$ (right column). The function $\Gamma_1 \sqcup \Gamma_2$ checks that one sequence is a suffix of the other and returns the longer one. The semantic function Π is described in text.

Now come two key ideas. Without special introduction they are easy to miss due to their simplicity. First, we curry the denotation: $\mathcal{E}[\tau_n, \dots, \tau_1 \vdash e : \tau] \in \mathcal{V}[\tau_n \rightarrow \dots \rightarrow \tau_1 \rightarrow \tau]$. Second, as the semantic domain $\mathcal{V}[\tau]$ we take the set of combinator expressions of type τ . (We shall soon see it is non-empty.) Alas, we have conflated closed and open terms: the combinator $I : \tau \rightarrow \tau$ may denote the closed term $\vdash \lambda z : \tau \rightarrow \tau$ as well as the open term $\tau \vdash z : \tau$. To distinguish the denotations of terms with the different number of free variables, we pair the combinator expression with I^2 .

Definition 2. *The denotation of a typed lambda-term $\mathcal{E}[\Gamma \vdash e : \tau]$ where Γ is τ_n, \dots, τ_1 is a tuple of an SKI term d of the type $\tau_n \rightarrow \dots \rightarrow \tau_1 \rightarrow \tau$, and the type sequence Γ . We write such tuple as $\Gamma \models d$.*

It follows that for a closed e , $\mathcal{E}[\vdash e : \tau]$ is a combinator expression of the type τ , which we take to be the result of our SKI translation³. The denotation $\mathcal{E}[\tau \vdash z : \tau]$ of the open term z is clearly $\tau \models I$: indeed, I , when applied to some d_0 , the one-component ‘environment’, reduces to that d_0 – the behavior

² As we will see in Sect. 6, it is enough to keep the length of Γ , that is, the number of free variables in a term.

³ It is natural to wish a denotation of an open term be non-divergent: if $\tau_n, \dots, \tau_1 \models d$ then d , until applied to n other terms, should have only a finite number of reductions, if any at all. The wish is already granted: in the present simply-typed calculus, all terms are (strongly) normalizing. We have to wait until Sect. 6 to say something non-trivial about termination.

expected of z . Likewise, we find that $\mathcal{E} [\tau, \sigma \vdash s z : \tau]$ is $\tau, \sigma \models K$: the combinator K , applied to d_1 and d_0 , the two-component environment, reduces to d_1 .

The right-hand side column of Fig. 6 describes the compositional computation of $\mathcal{E} [\Gamma \vdash e : \tau]$ in the form of inference rules. (EVar) was explained already. The (EAbs₀) rule says that if the function’s body is closed it is the constant function. (EAbs) amounts to η -conversion: if d is such that $(\tau_n, \dots, \tau_1, \sigma \models d)$, then $(d d_n \dots d_1)$ acts as a function: when applied to an argument $d_0 : \sigma$, that is, $d d_n \dots d_1 d_0$, it looks like d in the environment extended with d_0 . The (EW) rule states that weakening is the application of the K combinator.

The semantic function $(\Gamma' \models d') \amalg (\Gamma \models d)$ computes the application: that is, converts $(\tau_n, \dots, \tau_1 \models d')(\tau_m, \dots, \tau_1 \models d)$ to the form $\tau_{\max n m}, \dots, \tau_1 \models \bar{d}$ for some combinator expression \bar{d} . It is an exercise in combinatory logic. Its simplest (albeit not optimal) solution is to define \amalg by induction⁴: Let’s consider the case of the application of a closed term: $(\models d') \amalg (\tau_n, \dots, \tau_1 \models d)$. Our goal is to represent $d' (d d_n \dots d_1)$ as a combinator expression \bar{d} applied to d_n through d_1 . In the base case of $n = 0$, clearly \bar{d} is $d' d$. In the inductive case, the B reduction rule from Fig. 3 gives us $d' (d d_n \dots d_1) = (Bd')(d d_n \dots d_2) d_1$. Therefore, \bar{d} is the solution to the smaller instance of the problem: representing $(Bd')(d d_n \dots d_2)$ as $\bar{d} d_n \dots d_2$. In the general case of $(d' d_n \dots d_1)(d d_m \dots d_1)$ with $n \geq 1, m \geq 1$, the S reduction rule gives us $(S(d' d_n \dots d_2)) (d d_m \dots d_2) d_1$. Then we look for \bar{d}' such that $S(d' d_n \dots d_2) = \bar{d}' d_n \dots d_2$ (the earlier case of the closed-term application) and, finally, solve $(\bar{d}' d_n \dots d_2)(d d_m \dots d_2)$, which is the shorter version of the original problem. All in all, we obtain the following structurally recursive definition:

$$\begin{aligned} (\models d') \amalg (\models d) &= d' d \\ (\models d') \amalg (\tau_n, \dots, \tau_1 \models d) &= (\models Bd') \amalg (\tau_n, \dots, \tau_2 \models d) \\ (\tau_n, \dots, \tau_1 \models d') \amalg (\models d) &= (\models CCd) \amalg (\tau_n, \dots, \tau_2 \models d') \\ (\tau_n, \dots, \tau_1 \models d') \amalg (\tau_m, \dots, \tau_1 \models d) &= (\tau_n, \dots, \tau_2 \models (\models S) \amalg (\tau_n, \dots, \tau_2 \models d')) \amalg (\tau_m, \dots, \tau_2 \models d) \end{aligned}$$

Figure 7 shows the typing and semantic derivations for the running example: the flipped application $\lambda x. \lambda y. y x$. The typing derivation can be read as a proof, from the (Var) axioms down to the conclusion that the sample term has the type $\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$. Likewise, the semantic derivation produces the meaning of the term just as compositionally, from the (EVar) axioms down the chain of inference rules. The less-obvious step is the \amalg computation in the (EApp) rule:

$$\begin{aligned} (\alpha \rightarrow \beta \models I) \amalg (\alpha, \alpha \rightarrow \beta \models BKI) & \\ = (\models (\models S) \amalg (\models I)) \amalg (\alpha \models BKI) & \\ = (\models SI) \amalg (\alpha \models BKI) = B(SI)(BKI) & \end{aligned}$$

The overall result $B(SI)(BKI)$ is shorter than $S(S(KS)(KI))(S(KK)I)$ we obtained in Sect. 2 with the original Curry bracket abstraction – although far from being optimal. We describe the improvements in Sect. 3.1.

Figure 6, when read across, actually defines the translation process formally: each row of the figure gives the translation for the lambda-term of a particular

⁴ The optimal solution is described in Sect. 6.

$$\begin{array}{c}
\frac{\alpha \rightarrow \beta \vdash z : \alpha \rightarrow \beta}{\alpha, (\alpha \rightarrow \beta) \vdash z (sz) : \beta} \quad \frac{\alpha \vdash z : \alpha}{\alpha, \alpha \rightarrow \beta \vdash sz : \alpha} W \\
\frac{\alpha, (\alpha \rightarrow \beta) \vdash z (sz) : \beta}{\alpha \vdash \lambda z (sz) : (\alpha \rightarrow \beta) \rightarrow \beta} \\
\frac{\alpha \vdash \lambda z (sz) : (\alpha \rightarrow \beta) \rightarrow \beta}{\vdash \lambda \lambda z (sz) : \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta} \\
\frac{\alpha \models I}{\alpha, \alpha \rightarrow \beta \models BKI} EW \\
\frac{\alpha \rightarrow \beta \models I}{\alpha, (\alpha \rightarrow \beta) \models B(SI)(BKI)} \\
\frac{\alpha \models B(SI)(BKI)}{\models B(SI)(BKI)}
\end{array}$$

Fig. 7. The type and meaning derivations for the running example

form. For example, consider a term $\Gamma^+, \sigma \vdash se : \tau$. (Since this section deals with the Church-style calculus, each (sub)term comes annotated with its type and typing environment.) According to the second row of Fig. 6, we have to find the translation for $\Gamma^+ \vdash e : \tau$, which is $\Gamma^+ \models d$ for some combinator expression d . Next, we apply the (EW) rule. Lambda-abstraction is the only subtlety: whether to use (Abs/EAbs) or (Abs₀/EAbs₀) depends on the type environment. This formal process is implemented in OCaml, see Sect. 4.

To show correctness, we define the reverse translation $(d)_\Lambda$ from a combinator term d to a lambda-term, by substituting S, K, I, B, S combinators with the corresponding lambda-terms and treating combinator application as lambda-application.

Theorem 1 (Translation soundness/Semantics adequacy). *Let $\mathcal{E}[\tau_n, \dots, \tau_1 \vdash e : \tau]$ be $\tau_n, \dots, \tau_1 \models d$. Then $(d)_\Lambda s^{n-1}z \dots sz z =_{\beta\eta} e$*

This is the generalization of [10, Theorem 5.1.14] to open terms. The proof is the easy structural induction on the meaning derivation. The totality of the translation gives

Corollary 1 (Combinatorial Completeness). *Every, even open, lambda-term can be represented by a combinator term*

Or: every open term can be put in the form where all of its free variables are “on the right margin” – moreover, that form can be computed compositionally. So far, we have been dealing with the simply-typed calculus. Section 6 shows the general untyped case.

3.1 Lazy Weakening

The denotation $\tau_n, \dots, \tau_1 \models d$ says that a combinator d should be considered in an environment, of being applied to n other terms. For example, to understand the meaning of $\mathcal{E}[\tau, \sigma \vdash sz : \tau]$, which is $\tau, \sigma \models K$, we have to apply K to two other terms, d_2 and d_1 . This example also shows that some of these environment terms are ignored. Knowing what is ignored is useful: it leads to a better translation, as we are about to see.

To keep track of ignored context terms we mark them with the special “any type” - (analogous to the `_` type placeholder in, say, OCaml). Figure 8 shows the

$$\begin{array}{c}
 \Gamma \vdash e : \tau \qquad \mathcal{E}[\Gamma \vdash e : \tau] \\
 \hline
 \frac{\Gamma^+ \vdash e : \tau}{\Gamma^+, - \vdash s e : \tau} W \qquad \frac{\Gamma^+ \models d}{\Gamma^+, - \models d} EW \\
 \frac{\Gamma^+, - \vdash e : \tau}{\Gamma^+ \vdash \lambda e : \sigma \rightarrow \tau} Abs_1 \qquad \frac{\Gamma^+, - \models d}{\Gamma^+ \models (\models K) \amalg (\Gamma^+ \models d)} EAbs_1 \\
 \\
 \Gamma', - \sqcup \Gamma, - = (\Gamma' \sqcup \Gamma), - \qquad (\Gamma', - \models d') \amalg (\Gamma, - \models d) = (\Gamma' \models d') \amalg (\Gamma \models d) \\
 \Gamma', - \sqcup \Gamma, \tau = (\Gamma' \sqcup \Gamma), \tau \qquad (\Gamma', - \models d') \amalg (\Gamma, \tau \models d) = ((\models B) \amalg (\Gamma' \models d')) \amalg (\Gamma \models d) \\
 \Gamma', \tau \sqcup \Gamma, - = (\Gamma' \sqcup \Gamma), \tau \qquad (\Gamma', \tau \models d') \amalg (\Gamma, - \models d) = ((\models C) \amalg (\Gamma' \models d')) \amalg (\Gamma \models d)
 \end{array}$$

Fig. 8. The ‘weak lazy leftless’ type system and the denotational semantics. Shown are the differences from Fig. 6: the changed rule (W) and the new rule (Abs₁). The functions \sqcup and \amalg are also extended as shown. This system is presented in full in Sect. 4.

new type system and denotational semantics. The type system is clearly equivalent to that of Fig. 6. Now, the rule (EW) does nothing: ignored context terms are merely marked but ignored when computing the denotation. The real weakening is delayed until (EAbs₁): abstracting over an ignored variable gives the constant function. This rule corresponds to the so-called K-optimization, which ensures *full laziness* [7]. Within the bracket abstraction (Fig. 4) the optimizations is written [7, Sect. 16.2.1] as $\mathcal{A}_x[e] \mapsto Ke$ iff x is not free in e . We have accomplished it *without* needing to compute and search the set of free variables. (The sequence Γ , a part of the term denotation, is all we need to know about free variables.)

$$\begin{array}{c}
 \frac{\alpha \rightarrow \beta \vdash z : \alpha \rightarrow \beta}{\alpha, (\alpha \rightarrow \beta) \vdash z (sz) : \beta} \qquad \frac{\alpha \vdash z : \alpha}{\alpha, - \vdash sz : \alpha} W \\
 \hline
 \vdash \lambda \lambda z (sz) : \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \\
 \\
 \frac{\alpha \rightarrow \beta \models I}{\alpha, - \models I} EW \qquad \frac{\alpha \models I}{\alpha, - \models I} EW \\
 \hline
 \frac{\alpha, (\alpha \rightarrow \beta) \models B(CI)I}{\models B(CI)I}
 \end{array}$$

Fig. 9. The type and meaning derivations for the running example, in the weak lazy leftless system (the last two (Abs) steps are compressed)

The extended \amalg rules in Fig. 8 deal with applications of open terms, one of which (or both) ignore the first contextual term. For example, $(\tau_2, \tau_1 \models d') \amalg (\tau_2, - \models d)$ must be such \vec{d} that $\vec{d} d_2 d_1 = (d' d_2 d_1)(d d_2)$. The latter is equal to $C(d' d_2)(d d_2)d_1$ and hence the desired \vec{d} is the answer of $((\models C) \amalg (\tau_2 \models d')) \amalg (\tau_2 \models d)$. It is the so-called C optimization, which the new (EW) rule forces upon us. Figure 9 shows the typing and meaning derivations for the running example. Compared to Fig. 7, the translation result is both shorter and simpler

(avoiding the duplicating combinator S). Table 1 compares the two systems on more examples: tracking of ignored terms is truly a good optimization.

4 OCaml Implementation

The translation rules that previously have been written in mathematical notation are straightforward to turn into code, using the so-called tagless-final style [1, 5]. This section does so, taking OCaml as the implementation language. The reason to show the code in detail is actually theoretical: to present the translation as an algebra and to argue for its correctness.

Our OCaml code embeds both the lambda and SKI simply-typed calculi as DSLs, which are specified as OCaml module signatures `Lam` and `SKI`. The abstract type (γ, α) `repr` represents lambda-terms; it is parameterized by the term type α and the type environment γ . It might take time to realize that the `Lam` signature is the precise re-statement of the left column of Figs. 6 and 8, but in the OCaml-readable notation: `Lam` tells the syntax (with `$$` denoting an application) and the typing rules of the weak leftless system. (The operation $F' \sqcup F$ is done by unification during the type checking.)

```

module type Lam = sig
  type ( $\gamma, \alpha$ ) repr
  val z: ( $\alpha * \gamma, \alpha$ ) repr
  val s: ( $\beta * \gamma, \alpha$ ) repr  $\rightarrow$ 
        ( $_* (\beta * \gamma), \alpha$ ) repr
  val lam: ( $\alpha * \gamma, \beta$ ) repr  $\rightarrow$  ( $\gamma, \alpha \rightarrow \beta$ ) repr
  val ($$): ( $\gamma, \alpha \rightarrow \beta$ ) repr  $\rightarrow$ 
             ( $\gamma, \alpha$ ) repr  $\rightarrow$  ( $\gamma, \beta$ ) repr
end

module type SKI = sig
  type  $\alpha$  repr
  val kl: ( $\alpha \rightarrow \alpha$ ) repr
  val kK: ( $\alpha \rightarrow \beta \rightarrow \alpha$ ) repr
  val kS: (( $\alpha \rightarrow \beta \rightarrow \delta$ )  $\rightarrow$  ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $\alpha \rightarrow \delta$ ) repr
  val kB: (( $\alpha \rightarrow \beta$ )  $\rightarrow$  ( $\delta \rightarrow \alpha$ )  $\rightarrow$   $\delta \rightarrow \beta$ ) repr
  val kC: (( $\alpha \rightarrow \beta \rightarrow \delta$ )  $\rightarrow$  ( $\beta \rightarrow \alpha \rightarrow \delta$ )) repr
  val (!): ( $\alpha \rightarrow \beta$ ) repr  $\rightarrow$   $\alpha$  repr  $\rightarrow$   $\beta$  repr
end

```

In this notation, the running example is written as `lam (lam (z $$ (s z)))`. The typed SKI calculus is likewise defined by the signature `SKI` (where `!` is a SKI application). The `Lam` and `SKI` signatures clearly reveal the lambda and the combinator calculi to be algebras.

The accompanying code shows two implementations of the `SKI` signature, that is, two concrete SKI algebras. The carrier α `repr` of one algebra, `PSKI`, is set to be a `string`: it interprets every SKI expression as its printout. The other algebra is a SKI evaluator; its carrier is the set of OCaml values.

The lambda-to-SKI translation is also an algebra. It is an implementation of the `Lam` signature in terms of `SKI`:

```

module Conv(S:SKI) : Lam = struct
  type ( $\gamma, \alpha$ ) repr =
    | C:  $\alpha$  S.repr  $\rightarrow$  ( $\gamma, \alpha$ ) repr
    | N: ( $\gamma, \alpha \rightarrow \beta$ ) repr  $\rightarrow$  ( $\alpha * \gamma, \beta$ ) repr
    | W: ( $\gamma, \alpha$ ) repr  $\rightarrow$  ( $_* \gamma, \alpha$ ) repr

  let z: ( $\alpha * \gamma, \alpha$ ) repr = N (C S.kl) (*Var*)
  let s: ( $\beta * \gamma, \alpha$ ) repr  $\rightarrow$  ( $_* (\beta * \gamma), \alpha$ ) repr = fun e  $\rightarrow$  W e (*EW*)
end

```

```

let rec ( $\$$$ ): type g a b. (g,a→b) repr → (g,a) repr → (g,b) repr =
fun e1 e2 → match (e1,e2) with
| (W e1, W e2) → W (e1  $\$$$  e2)  (*( $se_1$ )( $se_2$ ) =  $s(e_1 e_2)$ *)
| (W e, C d) → W (e  $\$$$  C d)  (*( $se$ ) $d$  =  $s(ed)$ *)
| (C d, W e) → W (C d  $\$$$  e)
| (W e1, N e2) → N ((C S.kB)  $\$$$  e1  $\$$$  e2)
| (N e1, W e2) → N ((C S.kC)  $\$$$  e1  $\$$$  e2)
| (N e1, N e2) → N ((C S.kS)  $\$$$  e1  $\$$$  e2)
| (C d, N e) → N (C S.(kB $! d)  $\$$$  e)
| (N e, C d) → N (C S.(kC $! kC $! d)  $\$$$  e)
| (C d1, C d2) → C (S.(d1 $! d2))  (*closed term application*)

let lam: ( $\alpha$ * $\gamma$ , $\beta$ ) repr → ( $\gamma$ , $\alpha$ → $\beta$ ) repr = function
| C d → C S.(kK $! d)  (*Abs0*)
| N e → e  (*Abs*)
| W e → (C S.kK)  $\$$$  e  (*Abs1*)

let observe : (unit, $\alpha$ ) repr →  $\alpha$  S.repr = function (C d) → d
end

```

This implementation re-tells the right-column of Fig. 8, the bottom-up meaning computation, spelling out Π in full detail. The carrier is the GADT disjoint union of three sets: closed-term denotations (e.g., $\models I$ is written in OCaml as C S.kl), open-term denotations that ignore the context term (tagged with W) and general open-term denotations. If d of the OCaml type $(\gamma, \alpha \rightarrow \beta)$ repr represents $\Gamma \models d$ then $N\ d$: $(\alpha * \gamma, \beta)$ repr represents $\Gamma, \alpha \models d$. Interpreting the running example `lam (lam (z $\$$$ (s z)))` in this Conv(PSKI) algebra (with the PSKI implementation of SKI) gives the string “B(CI)I”.

Perhaps surprisingly, the implementation Conv makes a theoretical point. The type of `observe` reads as a proposition that a closed lambda-term translates to a SKI term of *the same type*. The type preservation is hence checked by the OCaml type checker. Furthermore, the OCaml compiler reports no inexhaustive pattern-match warnings: `observe` is hence total. The type-preservation of the translation gives, by free theorems, a certain degree of functional correctness. For example, the OCaml type checker assures that a lambda-term of the type $\alpha \rightarrow \beta \rightarrow \alpha$ is converted to a SKI term of the same type. Any term of that type (in a strongly normalizing calculus) have the same meaning. We see that not only the translation algorithm (Fig. 8) preserves the meaning but so does its implementation (the Conv functor).

4.1 The Eta-Optimization

The translation result for our running example, $B(CI)I$, shows the room for improvement. After all, B is the functional composition, of which I is the unit. Alternatively, $BdI \mapsto d$ is an η -reduction. We can implement this simplification as the second phase of the translation, in the standard tagless-final optimization framework⁵. It is more instructive to work it out into the translation itself.

⁵ <http://okmij.org/ftp/tagless-final/course/optimizations.html>.

Just as in Sect. 3.1, procrastination is the key: delay the introduction of the I combinator. Instead of using I for the denotation of z , we add a dedicated element V to our semantic domain:

```

type ( $\gamma, \alpha$ ) repr = | C:  $\alpha$  S.repr  $\rightarrow$  ( $\gamma, \alpha$ ) repr
                    | V: ( $\alpha * \gamma, \alpha$ ) repr
                    | N: ( $\gamma, \alpha \rightarrow \beta$ ) repr  $\rightarrow$  ( $\alpha * \gamma, \beta$ ) repr
                    | W: ( $\gamma, \alpha$ ) repr  $\rightarrow$  ( $_* \gamma, \alpha$ ) repr

```

with the corresponding changes to the semantic functions

```

let z: ( $\alpha * \gamma, \alpha$ ) repr = V
let lam: type a b g. (a * g, b) repr  $\rightarrow$  (g, a  $\rightarrow$  b) repr = function
  | V  $\rightarrow$  C S.kl
  ...

let rec ( $\$\$$ ): type g a b. (g, a  $\rightarrow$  b) repr  $\rightarrow$  (g, a) repr  $\rightarrow$  (g, b) repr =
fun e1 e2  $\rightarrow$  match (e1, e2) with
  | (W e, V)  $\rightarrow$  N e
  | (V, W e)  $\rightarrow$  N (C (S.(kC $! kl)) $\$ e)
  | (N e, V)  $\rightarrow$  N (C S.kS $\$ e $\$ C S.kl)
  | (V, N e)  $\rightarrow$  N (C S.(kS $! kl) $\$ e)
  | (C d, V)  $\rightarrow$  N (C d)
  | (V, C d)  $\rightarrow$  N (C S.(kC $! kl $! d))
  ...

```

Just like K in Sect. 3.1, I is introduced upon abstraction and in some cases upon applications. On the other hand, when applying a closed term d to V , the free variable is already at the right margin so the denotation becomes $\tau \models d$ with no extra combinators. With this optimization, the running example translates to mere CI , which is indeed the shortest combinator for the flipped application – quite an improvement over the naive translation $S(S(KS)(KI))(S(KK)I)$ in Sect. 1. The comparison Table 1 shows the current algorithm is by far the best: e.g., it translates the K combinator lambda-term to just K and the S combinator term to S . Yet the recursion in the $\$\$$ semantic function betrays non-linear complexity. We fix the problem in Sect. 6.

5 Compiling Real Programs

The simply-typed lambda-calculus considered so far is not even Turing-complete, let alone convenient. We want numbers, booleans, convenient conditionals, and general recursion. Fortunately, all these features are easy to add, as constants (assuming a non-strict evaluation strategy) of appropriate types. Below is an example, borrowed from [7, Sect. 16.2.6]. It is the lambda-term for finding the greatest common divisor of two integers a and b (with $b \leq a$) using Euclid’s algorithm, written in the extended calculus embedded in OCaml:

```

let gcd = fix $\$ lam (lam (lam
  (let self = s (s z) and a = s z and b = z in
    if_ $\$ (eq $\$ int 0 $\$ b) $\$ a $\$ (self $\$ b $\$ (rem $\$ a $\$ b))))))

```

(The let-expression of the host language (OCaml) is the free syntax sugar that makes the term more readable.) Its translation to SKI (also extended with combinators such as Y , IF , Rem and 0) is

$$Y(B(S(BS(C(B IF (= 0)))))(CC Rem (BBS)))$$

The algorithm from Sect. 4.1 was used as it is, treating fix, if_, etc. constants as primitive combinators. The result is compact and can be shown with no ‘cheating’ ([7, Sect. 16.2.6] translated only the function’s body, without the recursive knot).

6 Linear algorithm

This section describes the time- and space-linear translation algorithm, in the general case of the untyped lambda-calculus (which can be backported to the typed case). We stress that for clarity the algorithm is based on the simpler Fig. 6 rather than the optimized Fig. 8 (and hence has room for improvement).

However odd, we continue to use ‘type derivations’, whose judgments omit types, and represent the ‘type’ environment Γ by its length, the natural number: e.g., $1 \vdash z$ (with the denotation $1 \models I$). The zero length is omitted. The unitype system and the corresponding denotation rules are presented in Fig. 10, which is the trivial simplification of Fig. 6.

The only interesting part is the new definition of the semantic function Π , which computes the denotation of application. Its defining requirement is

$$(n \models d') \Pi (m \models d) = \bar{d} \quad \text{iff} \quad (d' d_n \dots d_1)(d d_m \dots d_1) = \bar{d} d_{\max n m} \dots d_1$$

for some terms $d_{\max n m}, \dots, d_1$. Previously, in Sect. 3, such \bar{d} was computed by induction on $\max n m$. (The two applied terms share the $\min n m$ suffix of their environment, which is easy to see from Fig. 6.) In contrast, Fig. 10 defines Π in terms of so-called bulk combinators B_n , C_n and S_n , without any recursion. The bulk combinators (whose reductions and definitions in terms of S and K are collected in Fig. 11) are specifically designed to satisfy the defining requirement of Π . For example:

$$\begin{aligned} (d' d_n \dots d_1)(d d_n \dots d_1) &= S_n d' d d_n \dots d_1 \\ &\equiv ((n \models d') \Pi (n \models d)) d_n \dots d_1 \\ (d' d_{n+k} \dots d_1)(d d_n \dots d_1) &= S_n (d' d_{n+k} \dots d_{n+1}) d d_n \dots d_1 \\ &= (B_k S_n d' d_{n+k} \dots d_{n+1}) d d_n \dots d_1 \\ &= C_k (B_k S_n d') d d_{n+k} \dots d_{n+1} d_n \dots d_1 \\ &\equiv ((n+k \models d') \Pi (n \models d)) d_{n+k} \dots d_1 \quad k > 1 \end{aligned}$$

The bulk B_n , C_n and S_n combinators, like the ordinary B , C and S , take two terms d' and d and then $n \geq 1$ more terms and distribute the latter across the first two. The bulk combinators are thus the generalization of ordinary ones, to

$n \vdash e$	$\mathcal{E}[n \vdash e]$
$\frac{}{1 \vdash z} \text{Var}$	$\frac{}{1 \models I} \text{EVar}$
$\frac{n+1 \vdash e}{n+2 \vdash s e} W$	$\frac{n+1 \models d}{n+2 \models (\models K) \amalg (n+1 \models d)} EW$
$\frac{\vdash e}{\vdash \lambda e} \text{Abs}_0$	$\frac{\models d}{\models K d} \text{EAbs}_0$
$\frac{n+1 \vdash e}{n \vdash \lambda e} \text{Abs}$	$\frac{n+1 \models d}{n \models d} \text{EAbs}$
$\frac{n \vdash e_1 \quad m \vdash e_2}{\max n m \vdash e_1 e_2} \text{App}$	$\frac{n \models d_1 \quad m \models d_2}{\max n m \models (n \models d_1) \amalg (m \models d_2)} \text{EApp}$
$(\models d_1) \amalg (\models d_2) = d_1 d_2$	
$(\models d_1) \amalg (n \models d_2) = B_n d_1 d_2$	
$(n \models d_1) \amalg (\models d_2) = C_n d_1 d_2$	
$(n \models d_1) \amalg (n \models d_2) = S_n d_1 d_2$	
$(n \models d_1) \amalg (m \models d_2) = B_{m-n}(S_n d_1) d_2$	if $n < m$
$(n \models d_1) \amalg (m \models d_2) = C_{n-m}(B_{n-m} S_m d_1) d_2$	if $n > m$

Fig. 10. The untyped system and the denotational semantics

distribute several terms ‘in bulk’. (The bulk combinators are typeable and hence usable also in the typed case.)

The untyped combinator calculus permits divergent terms. The denotation of the closed $(\lambda z z)(\lambda z z)$ comes out as $(SII)(SII)$; the divergent combinator term as denotation is natural and expected. One may wish, however, denotations of open terms be convergent. To this end, one may read $n \models d$ as if it were $n \models I_n d$. The (assumed) I_n combinator ensures no reductions taking place until the full environment is supplied.

$B' d f g x \rightsquigarrow d f (gx)$	$B' = BB$
$C' d f g x \rightsquigarrow d (fx) g$	$C' = B(BC)B$
$S' d f g x \rightsquigarrow d (fx) (gx)$	$S' = B(BS)B$
$B_n f g x_n \dots x_1 \rightsquigarrow f (g x_n \dots x_1)$	$B_n = \text{times}_{n-1} B' B$
$C_n f g x_n \dots x_1 \rightsquigarrow (f x_n \dots x_1) g$	$C_n = \text{times}_{n-1} C' C$
$S_n f g x_n \dots x_1 \rightsquigarrow (f x_n \dots x_1) (g x_n \dots x_1)$	$S_n = \text{times}_{n-1} S' S$
$I_n f x_n \dots x_1 \rightsquigarrow f x_n \dots x_1$	$I_n = B_n I$ (we set I_0 to be I)

Fig. 11. Primed (director) and bulk combinators: reductions and SK definitions. The n -times application of d to d' is denoted as $\text{times}_n d d'$.

In the earlier approaches, the size of the translation result has to be, in the worst case, at least quadratic in the size of the input term. This is easy to see on the worst term $\lambda x_1 \dots \lambda x_n. x_n \dots x_1$ from [7]. Its body has n variables each requiring a different component from the environment, and $n - 1$ applications. Distributing one environment variable across an application requires one combinator. The bulk combinators distribute several variables in bulk, hence we expect improvement.

Let us analyze the time and space complexity of the translation. The complexity measure is the size of the input lambda term, or the number of its constructors: z , s , λ and the application. Alternatively, this is the size of the term's typing derivation, as each constructor corresponds to a rule in the derivation. If we regard B_n , C_n and S_n as pre-computed (see below) and take as the cost metric the number of combinator applications, we see from Fig. 10 that each (typing or semantic) rule contributes at most 4 combinator applications (the worst case is the application of $n \vdash e_1$ to $m \vdash e_2$ where $n > m$). We stress that Π is no longer recursive. With the reasonable representation of combinator terms (e.g., as trees) an application takes fixed amount of time. Thus, the time complexity is linear. The size of the created term is also proportional to the number of applications (i.e., the internal nodes of the binary tree of the term). Therefore, the space complexity – the size of the result – is also linear in the size of the input term. The last three rows of Table 1 lists the worst-case terms of increasing size. The last column clearly shows the linear size increase for the present translation. Overall, the experience so far (including the worst-case examples of [6, 7]) shows the number of combinators in the translation result stays within $1.5 \times$ the size of the input term.

The linear time- and space complexity may seem surprising. To intuitively understand it, let's apply a grossly simplified translation to $\lambda x_1 \dots \lambda x_n. e_1 e_2$. First we distribute the n -component environment to both e_1 and e_2 with the bulk S_n . That costs us only one combinator. If, say, e_1 is a variable x_i , it has to project one component from the environment. Realizing the projection may take $\Theta(n - i)$ combinators. However, in De Bruijn notation, x_i is encoded as $s^{n-i} z$, whose size $n - i + 1$ pays for the projection combinators. If e_1 happens to be an application, we again use S_n to distribute the environment to both applicands. It costs us one combinator, which is paid by the application (which adds one to the size of the input term). The real translation uses B_n and C_n to avoid distributing unused prefixes of the environment.

We have assumed that B_n , C_n and S_n have been precomputed (as they would be in practice). If not, we have to compute them: scan the input term to determine its size n and compute and store the three sequences of bulk combinators, of n elements each. Each sequence, say, B_i , $1 \leq i \leq n$, is built by iteration, applying the primed combinator such as B' to the previous element of the sequence. All in all, the required time and space is linear in n .

We regard arithmetic operations (such as integer comparison, subtraction, etc) and array dereference (to fetch a pre-computed bulk combinator) taking constant time – common in analyses of data structures. (All numbers in our

algorithm are bound by the size of the input term. Therefore, if the size of a term does not fit within a machine register, the term does not fit into memory. Processing such terms has a very different cost model, so the traditional complexity analysis becomes pointless.)

7 Related Work

The bracket abstraction is a classical, textbook algorithm, with many descriptions and explanations and blogs⁶; we have already mentioned [7]. The algorithm and its many variations are syntactic, with typical optimization side-conditions of the form $x \notin FV(e)$. In our semantic approach, the type environment, being part of the denotation, is all we need to know about free variables. We never search through it.

The most recent work on combinator translation [9] involves so-called director strings [7, Sect. 16.3], which tell for each application node which parameters should go left or right or both ways. The bulk combinators in Fig. 11 do a similar job, but in bulk: directing whole environments rather than single variables.

Table 1. Translation of the examples using different methods and optimizations.

Lambda-term	Size	Figure 6	Figure 8	Section 4.1	Section 6
$\lambda\lambda z$	3	KI	KI	KI	KI
$\lambda\lambda sz$	4	BKI	BKI	K	BKI
$\lambda\lambda sz z$	6	CCI (BS(BKI))	CCI(BBI)	I	C(BS(BK))I
$\lambda\lambda z sz$	6	B(SI)(BKI)	B(CI)I	CI	B(SI)(BKI)
$\lambda\lambda\lambda z s^2 z$	8	B(B(SI)) (B(BK)(BKI))	BK(B(CI)I)	BK(CI)	B ₂ (SI) (B ₂ K(BKI))
$\lambda\lambda\lambda(\lambda z) s^2 z$	9	B(B(BI)) (B(BK)(BKI))	BK(BK(BII))	BK(BKI) [‡]	B ₃ (B ₂ K(BKI))
$\lambda\lambda\lambda(s^2 z z) (s z z)$	13	CC(CCI(BS(BKI))) (BS(B(BS)(B(CCI)(B(BS) (B(BK)(BKI))))))	CC(CCI(BBI)) (BB(BS (CCI(BBI))))	S	C(BS ₂ (C ₂ (B ₂ S(B ₂ K (BKI))))I)) (C(BS(BKI))I)
The worst-case family for the combinator translation					
$\lambda\lambda z sz$	6	B(SI)(BKI)	B(CI)I	CI	B(SI)(BKI)
$\lambda\lambda\lambda z sz s^2 z$	11	B(S(BS (B(SI)(BKI)))) (B(BK)(BKI))	B(C(BC (B(CI)I))I)	C(BC(CI))	B(S ₂ (B(SI)(BKI)) (B ₂ K(BKI))
$\lambda\lambda\lambda\lambda z sz s^2 z s^3 z$	17	B(S(BS(B(BS (B(S(BS(B(SI)(BKI)))) (B(BK)(BKI)))))) (B(B(BK)) (B(BK)(BKI))))	B(C(BC(B(BC (B(C(BC (B(CI)I))))))I)	C(BC(B(BC (C(BC(CI))))))	B(S ₃ (B(S ₂ (B(SI)(BKI)) (B ₂ K(BKI)) (B ₃ K(B ₂ K (BKI))

[‡]The unoptimized (BKI) comes from a redex: the input term is not normal.

Hughes' supercombinators [3] is the alternative combinator-based implementation strategy for functional languages, tightly connected to lambda-lifting. Unlike supercombinators, which are program-specific, our bulk combinators are

⁶ It is worth pointing out one, comprehensive web page: <http://www.cantab.net/users/antoni.diller/brackets/intro.html>.

general-purpose and can be pre-computed once and for all (or even wired into hardware). The supercombinators and the SKI translation are extensively compared in [7, Sect. 16.4], including the performance: the SKI translation time and the size of the resulting combinator term are worst-case quadratic in the size n of the (*not* De Bruijn-indexed) input lambda-term (although the typical complexity is $O(n \log n)$). In contrast, Hughes supercombinator process has the worst-case size complexity of $O(n \log n)$, but is often linear.

Our bulk combinators B_n, C_n, S_n are the same as Noshita's [6] $\bar{B}_n, \bar{C}_n, \bar{S}_n$ (but not his B_n, C_n, S_n), although introduced and used differently. (Like [6], we take each bulk combinator reference – a pointer to the pre-computed combinator sequence – to take constant space.) Noshita's combinator translation algorithm is the extension of Turner's and is syntactic. Noshita proves the $O(n \log n)$ upper-bound on the translation size; there is no claims about time complexity. Comparing ours and Noshita's approaches are difficult: not only the translation algorithms differ and give different (but equivalent) results; our calculi and input size measures also differ. Ours is lambda-calculus with De Bruijn indexes (since we also work with the typed calculus, s has to be a constructor and is counted as such). Noshita's input are binary trees with constants and named variables at the leaves – but no lambdas: Noshita only deals with supercombinators.

8 Conclusions

We have presented a semantic approach to translating lambda-terms to SKI combinators: the translation is a compositional computation of the meaning of a term. The key ideas are the choice of the semantic domain (the set of combinators) and the representation of open terms. Our presentation has stressed the parallel between type- and meaning derivations. We have demonstrated how easy, 'natural' it is to introduce various optimizations – leading all the way to the time- and space- linear translation algorithm.

The semantic approach easily extends to untyped calculus and to real programs with integers, conditionals, fixpoint, etc.

The linear translation algorithm has all the attractiveness of the supercombinator approach, but using general-purpose combinators, which can be pre-computed or even wired-in. Perhaps combinators as the compilation target of real functional languages deserve a second look.

Acknowledgments. I thank Yuki Yoshi Kameyama for his challenge to write the SK conversion in the tagless-final style, and helpful discussions. I am very grateful to Doaitse Swierstra, Fritz Henglein and Noam Zeilberger for many helpful comments and discussions. Numerous suggestions by anonymous reviewers have greatly helped improve the presentation.

References

1. Carette, J., Kiselyov, O., Shan, C.C.: Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages. *J. Funct. Program.* **19**(5), 509–543 (2009)
2. Curry, H.B., Feys, R.: *Combinatory Logic*. North-Holland, Amsterdam (1958)
3. Hughes, R.J.M.: Super combinators: a new implementation method for applicative languages. In: *Symposium on LISP and Functional Programming*, pp. 1–10. ACM, August 1982
4. Joy, M.S., Rayward-Smith, V.J., Burton, F.W.: Efficient combinator code. *Comput. Lang.* **10**(3/4), 211–224 (1985)
5. Kiselyov, O.: Typed Tagless Final Interpreters. In: Gibbons, J. (ed.) *Generic and Indexed Programming*. LNCS, vol. 7470, pp. 130–174. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32202-0_3
6. Noshita, K.: Translation of Turner combinators in $O(n \log n)$ space. *Inf. Process. Lett.* **20**(2), 71–74 (1985)
7. Peyton Jones, S.: *The Implementation of Functional Programming Languages*. Prentice Hall, Upper Saddle River, January 1987. <https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/>
8. Schönfinkel, M.: Über die Bausteine der mathematischen Logik. *Math. Ann.* **92**(3), 305–316 (1924)
9. Sinot, F.R.: Director strings revisited: a generic approach to the efficient representation of free variables in higher-order rewriting. *J. Log. Comput.* **15**(2), 201–218 (2005)
10. Sørensen, M.H., Urzyczyn, P.: *Lectures on the Curry-Howard isomorphism*. Technical report 98/14 (TOPPS note D-368), DIKU, Copenhagen (1998)
11. Stoye, W.R.: *The implementation of functional languages using custom hardware*. Ph.D. thesis, Computer Laboratory, University of Cambridge, December 1985. <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-81.pdf>
12. Turner, D.A.: Another algorithm for bracket abstraction. *J. Symb. Log.* **44**(2), 267–270 (1979)
13. Turner, D.A.: A new implementation technique for applicative languages. *Softw.-Pract. Exp.* **9**, 31–49 (1979)



Program Extraction for Mutable Arrays

Kazuhiko Sakaguchi^(✉)

University of Tsukuba, Tsukuba, Japan
sakaguchi@coins.tsukuba.ac.jp

Abstract. We present a mutable array programming library for the Coq proof assistant which enables simple reasoning method based on Ssreflect/Mathematical Components, and extractions of the efficient OCaml programs using in-place updates. To refine the performance of extracted programs, we improved the extraction plugin of Coq. The improvements are based on trivial transformations for purely functional programs and reduce the construction and destruction costs of (co)inductive objects, and function call costs effectively. As a concrete application for our library and the improved extraction plugin, we provide efficient implementations, proofs, and benchmarks of two algorithms: the union-find data structure and the quicksort algorithm.

1 Introduction

The program extraction mechanism [7–9, 15] of the Coq proof assistant [20] is a code generation method for obtaining certified functional programs from constructive formal proofs and definitions by eliminating the non-informative part and widely used for developments of high-reliability software. For example, the CompCert project [6] uses Coq and its program extraction mechanism for obtaining a formally verified and executable C compiler which guarantees the correctness of the translation.

Verification and code generation (including extraction) of programs with side effects are important issues to apply proof assistants to realistic software developments. Particularly, in-place updates of mutable objects are important to increase the efficiency of programs. There are many recent studies to support writing, reasoning about, and generating programs with side effects in proof assistants, e.g., Coq (Ynot) [13], Isabelle/HOL [2], Idris [1], and F* [17]. Ynot is such a representative Coq library which can handle various kind of side-effects, e.g., accessing reference cells, non-termination, throwing/catching exceptions, and I/O. Ynot is based on an axiomatic extension for Coq called Hoare Type Theory (HTT) [12] and supports reasoning with separation logic [14] which are good at its expressiveness, but not good at reducing proof burden in Coq.

This study establishes a novel, lightweight, and axiom-free method for verification and extraction of efficient programs using mutable arrays. Our library supports a simple monadic DSL for mutable array programming and powerful and simple reasoning method, and which is achieved by focusing only on mutable

arrays and doing away with more side-effects such as reference cells and local states. Our contribution consists of three parts:

- In Sect. 3, we define a state monad specialized for mutable array programming—the array state monad—and give two interpretations of it for reasoning and program extraction. The former interpretation is defined in a purely functional way with building blocks taken from the Ssreflect/Mathematical Components (MathComp) library [11, 21] and makes it possible to reduce the reasoning tasks on effectful programs to those on purely functional programs. The latter interpretation enables extraction of efficient effectful programs and provides encapsulation function like `runST` of state threads [5] which corresponds to the interpretation function in the former interpretation. The encapsulation mechanism converts effectful functions written by the array state monad to referential transparent functions and it also enables encapsulation of proofs.
- In Sect. 4, we present two new optimization techniques for the program extraction plugin. The optimizations are based on well-known transformations of purely functional programs, but effectively reduce the execution time of programs extracted with our library. More generally, the optimizations are especially effective for two cases: 1. proofs using mathematical structures and its theories provided by the MathComp library and 2. programs using monads that have functional types, e.g., State, Reader, and Continuation monads.
- In Sect. 5, we demonstrate elegant formalization techniques for programs using mutable arrays, and efficiency of the extracted programs using our library and the improved extraction, through the two applications—the union–find data structure and the quicksort algorithm.

Our formalization of the quicksort algorithm uses the theory of permutations provided by the `perm` library of MathComp, and we show that some properties of permutations are also helpful for the reasoning of the quicksort algorithm. We also show benchmark results of the applications, and it indicates that performances of extracted programs using our library and the improved extraction are comparable to handwritten OCaml implementations of same algorithms and much better than purely functional implementations of the same kind of algorithms.

The source code of our library, the improved extraction plugin, case studies, benchmark scripts and patches for existing libraries are available at:

<https://github.com/pi8027/efficient-finfun>.

2 Finite Types and Finite Functions in Coq

This section briefly introduces the `fintype` and `finfun` libraries from the MathComp library with some modifications. The former provides an interface for types with finitely many elements—*finite types*. The latter provides a type of functions with finite domains—*finite functions* or *finfuns*—which is used as a

representation of arrays in this paper and relies on the former part. Key definitions and lemmas of the both libraries are listed in Table 1 and described below. In our previous work [16], we modified these libraries to improve the efficiency of code extracted from proofs.

2.1 A Finite Type Library—`finType`

A finite type is a type with finitely many elements. The `finType` library provides definitions of a class of finite types (`finType`) and its basic operations. The class of finite types is defined as a canonical structure [10] which contains a type and a witness of its finiteness. In the original `finType` library, such finiteness of a type `T` is characterized by a duplicate-free enumeration of elements of `T`. In the modified one, it is recharacterized by a pair of the natural cardinal number `c` and a bijection between `T` and a finite ordinal type `'I_c = {0, ..., c - 1}`. The bijection is given by a pair of an encoding function of type `T → 'I_c` and a decoding function of type `'I_c → T`.

The two most basic operations on finite types are the enumeration (`enum`) and the cardinality (`#|_l`) of a subset of a finite type. We also provide accessors for the cardinal number and the bijection used by the new characterization of finiteness. The former is `$|_l` notation, and the latter is `fin_encode` and `fin_decode` functions. For any `T : finType`, `$|T|` is equal to `#|T|` and more efficiently computable than `#|T|`.

Many canonical `finType` instances are given by the `MathComp` library: `unit`, `bool`, finite ordinals `'I_n`, `option`, `sum`, `prod`, finite functions with a finite codomain, finite subsets `{set T}`, symmetric groups `{perm T}`, and more.

2.2 A Finite Function Library—`finfun`

The `finfun` library provides definitions of a type of finfun (`{ffun T → U}`) and its basic operations. Finfun from `T` to `U` is defined as `#|T|` tuples of `U`. Tuples are size-fixed lists defined in the tuple library, and easily translated to arrays in extracted programs by the `Extract Inductive` command.

`Inductive finfun_type (T : finType) (U : Type) := Finfun of #|T|.tuple U.`

The two most basic operations of finfun are the application (`fun_of_fin`) and the construction from CiC functions (`finfun`). The application `fun_of_fin f i` is the `(fin_encode i)`-th element of the underlying tuple of `f`. The construction `finfun f` is the finfun extensionally equal to `f`, and the underlying tuple of it associates `f (fin_encode i)` for each `i`-th element. Both of them can be efficiently computable by using `Array.get` and `Array.init` functions respectively in extracted OCaml programs.

The application `fun_of_fin f i` can be expressed as `f i` because `fun_of_fin` is the coercion from finfun to CiC functions. The `finfun` library also provides constructor notations `[ffun i ⇒ e]` and `[ffun ⇒ e]` that are equivalent to `finfun (fun i ⇒ e)` and `finfun (fun _ ⇒ e)` respectively.

Table 1. Key definitions and lemmas in the modified fintype and finfun libraries

Coq judgement	Informal semantics
$T : \text{finType}$	T is a type with finitely many elements
$T : \text{finType} \vdash \text{Finite.sort } T : \text{Type}$	<i>coercion</i> from finType to Type
$T : \text{Type} \vdash [\text{finType of } T] : \text{finType}$	<i>canonical finType</i> instance of T
$n : \text{nat} \vdash 'I_n : \text{Type}$	type of natural numbers $\{0, \dots, n-1\}$ (finite ordinal type)
$n : \text{nat}, i : 'I_n \vdash \text{nat_of_ord } i : \text{nat}$	<i>coercion</i> from finite ordinal type to nat
$T : \text{finType}, pT : \text{predType } T, p : pT \vdash \text{enum } p : \text{seq } T$	enumeration of the subset p of T
$T : \text{finType}, pT : \text{predType } T, p : pT \vdash \# p : \text{nat}$	cardinality of the subset p of T
$T : \text{finType} \vdash \$ T : \text{nat}$	cardinality of the finite type T
$T : \text{finType}, x : T \vdash \text{fin_encode } x : 'I_{\# T }$	} bijection between T and $'I_{\# T }$
$T : \text{finType}, i : 'I_{\# T } \vdash \text{fin_decode } i : T$	
$T : \text{finType}, x : T \vdash \text{fin_encodeK } x : \text{fin_decode } (\text{fin_encode } x) = x$	fin_decode is a left inverse of fin_encode
$T : \text{finType}, i : 'I_{\# T } \vdash \text{fin_decodeK } i : \text{fin_encode } (\text{fin_decode } i) = i$	fin_encode is a left inverse of fin_decode
$T : \text{finType}, U : \text{Type} \vdash \{\text{ffun } T \rightarrow U\} : \text{Type}$	ffun from T to U
$T : \text{finType}, U : \text{Type}, \vdash \text{fun_of_fin } f : i : U$	image of i under f (<i>coercion</i> from ffun to CiC functions)
$T : \text{finType}, U : \text{Type}, f : T \rightarrow U \vdash \text{finfun } f : \{\text{ffun } T \rightarrow U\}$	finfun such that extensionally equal to the CiC function f
$T : \text{finType}, U : \text{Type}, \vdash \text{ffunP } f : f = 1 \text{ } g \leftrightarrow f = g$	functional extensionality of ffun ^a
$T : \text{finType}, U : \text{Type}, \vdash \text{ffunE } f : x : \text{ffun } f \rightarrow U, x : T \vdash \text{fun_of_fin } (\text{finfun } f) : x = f \text{ } x$	unfolding lemma for application of ffun

^a $=1$ means an extensional equality with arity 1.

3 Representing Mutable Arrays in Coq

The state monad [22] is useful to represent computations with states in purely functional languages such as Haskell and Coq. *Actions* of the state monad have types of the form $S \rightarrow S \times A$ where S is a state type and A is a return type. They take the initial state as its arguments, and returns the result of the type A paired with the final state. To represent various computations with mutable arrays, we need an *array state monad* that hold two conditions: (C1) it can handle multi-dimensional and multiple mutable arrays, and (C2) it never needs copy operations on arrays.

The former part of (C1)—handling multi-dimensional arrays—is achieved naturally by representing arrays by finfun, because the finfun of type $\{\text{ffun } I_1 \times \dots \times I_n \rightarrow A\}$ correspond to the multi-dimensional arrays of A indexed by I_1, \dots, I_n .

Monad transformers are well-known methods to compose monadic effects, and the state monad transformer seem to be suitable for solving the latter part of (C1)—handling multiple arrays. However, if we allow one to compose the array state monad and other monads, the condition (C2) does not hold. For example, the actions of the monad that is a composition of the state monad transformer and the list monad have a type $S \rightarrow \text{list}(S \times A)$, and it needs to copy the state on each branch of computation. Therefore, the array state monad should be defined in a more refined way.

We solve the above problem by defining the array state monad as an inductive data type that has a restricted set of primitive operations shown below:

Definition `Sign : Type := seq (finType * Type)`.

Implicit Types `(I J K : finType) (sig : Sign)`.

Inductive `AState : Sign → Type → Type :=`
`| astate_ret_ : ∀{sig} {A : Type}, A → AState sig A`
`| astate_bind_ :`
`∀{sig} {A B : Type}, AState sig A → (A → AState sig B) → AState sig B`
`| astate_lift_ :`
`∀{I} {T : Type} {sig} {A : Type}, AState sig A → AState ((I, T) :: sig) A`
`| astate_GET_ : ∀{I} {T : Type} {sig}, 'I_#|I| → AState ((I, T) :: sig) T`
`| astate_SET_ :`
`∀{I} {T : Type} {sig}, 'I_#|I| → T → AState ((I, T) :: sig) unit.`

`AState [:: (I1, T1); ...; (In, Tn)] A` is a type of array state monad actions with I_1, \dots, I_n indexed mutable arrays of T_1, \dots, T_n respectively and the return type A . The first argument of `AState` is called a *signature* and indicates types of mutable arrays. Let Σ be a metavariable of signatures, and Σ_i means i -th element of the signature Σ . We refer to the array corresponds to the Σ_i as the *i -th array (of the signature Σ)*.

Each constructor of `AState` corresponds to return, bind, lift, get and set operators. The lift operator can lift array state monad actions of a signature Σ to that with a signature $(I, T) :: \Sigma$, and lifted actions does not affect the first array. Get and set operators can only access the first array. The lift operator

is necessary to get and set element of an array after the second, and it is also useful for modular programming.

We also define aliases for all constructors of `AState` to avoid the construction and destruction costs of tuples in extracted OCaml programs¹, e.g.:

```
Definition astate_ret {sig A} a := @astate_ret_ sig A a.
```

Primitive get and set operators take an index of type `'I_#|I|`. Indices of type `I` are also applicable by using the encoding function.

```
Notation astate_get i := (astate_GET (fin_encode i)).
```

```
Notation astate_set i x := (astate_SET (fin_encode i) x).
```

Programs represented by `AState` values cannot run directly. We give an interpretation for the array state monad by a translation to the functions of type $S \rightarrow S \times A$, where S is a type of mutable arrays defined as follows:

```
Fixpoint states_AState sig : Type :=
  if sig is (Ix, T) :: sig' then states_AState sig' * {ffun Ix → T} else unit.
```

`states_AState` takes a signature $[(I_1, T_1); \dots; (I_n, T_n)]$, and returns a Cartesian product of all types of arrays in the signature: $\text{unit} \times \{\text{ffun } I_n \rightarrow T_n\} \times \dots \times \{\text{ffun } I_1 \rightarrow T_1\}$. We choose this order of types to omit parenthesis, because the \times and (\cdot, \cdot) operators have left associativity in Coq.

The translation is defined as follows:

```
Definition ffun_set
  (I : finType) (T : Type) (i : I) (x : T) (f : {ffun I → T}) :=
  [ffun j ⇒ if j == i then x else f j].
```

```
Definition runt_AState sig (A : Type) : Type :=
  states_AState sig → states_AState sig * A.
```

```
Definition run_AState : ∀sig A, AState sig A → runt_AState sig A :=
  @AState_rect (fun sig A _ ⇒ runt_AState sig A)
  (* return *) (fun _ _ a s ⇒ (s, a))
  (* bind *) (fun _ _ _ f _ g s ⇒ let (s', a) := f s in g a s')
  (* lift *) (fun _ _ _ _ f '(s, a) ⇒ let (s', x) := f s in (s', a, x))
  (* get *) (fun _ _ _ i s ⇒ (s, s.2 (fin_decode i)))
  (* set *) (fun _ _ _ i x '(s, a) ⇒ (s, ffun_set (fin_decode i) x a, tt)).
```

`ffun_set` is a pure set function for finfun. It takes an index $i : T$, a value $x : A$ and a finfun $f : \{\text{ffun } T \rightarrow A\}$, and returns a new finfun f' which is equal to f except that the i -th element is changed to x . `run_AState` is a interpretation for the array state monad which is inductively defined on `AState` values.

3.1 Program Extraction for the Array State Monad

This section provides a method to extract efficient stateful OCaml programs from Coq proofs which use the array state monad. In another point of view, we

¹ In OCaml programs, arguments of constructors are parenthesized and comma separated. If a constructor is replaced with some function by the `Extraction Inductive` command, arguments of the constructor are interpreted as tuples.

give an another interpretation for array state monad by the OCaml program extraction.

In stateful settings, state propagation can be achieved by in-place updates instead of state monad style propagation, and moreover it is not needed to return a new state in each action. We defined the array state monad as an inductive data type only because to restrict primitive operations and its case analysis is never used except for `run_AState`. Thus we interpret array state monad actions as OCaml functions which take states and return its result by the `Extract Inductive` command:

Definition `runt_AState_ sig (A : Type) : Type := states_AState sig → A.`

```
Extract Inductive AState ⇒ "runt_AState_"
  [(* return *) " (fun a s -> a)"
   (* bind *)   " (fun (f, g) s -> let r = f s in g r s)"
   (* lift *)   " (fun f s -> let (ss, _) = Obj.magic s in f ss)"
   (* get *)    " (fun i s -> let (_, s1) = Obj.magic s in s1.(i))"
   (* set *)    " (fun (i, x) s -> let (_, s1) = Obj.magic s in s1.(i) <- x)"]
  "(* It is not permitted to use AState_rect in extracted code. *)".
```

We also give same realizations of aliases for the constructors of `AState`:

```
Extract Inlined Constant astate_ret ⇒ "(fun a s -> a)".
Extract Inlined Constant astate_bind ⇒ "(fun f g s -> let r = f s in g r s)".
...

```

Finally, we provide an encapsulation function for the array state monad as a realization of the `run_AState` function:

```
Extract Constant run_AState ⇒
  "(fun sign f s ->
   let rec copy sign s =
     match sign with
     | [] -> Obj.magic ()
     | _ :: sign' -> let (s', a) = Obj.magic s in
                     Obj.magic (copy sign' s', Array.copy a) in
   let s' = copy sign s in
   let r = Obj.magic f s' in (s', r))".
```

The encapsulation is achieved by duplicating all the input (initial) arrays by `Array.copy` and using the copied arrays in the execution of effectful actions. As a result, the range of in-place updates is limited to the copied arrays and it never affects outside of the `run_AState`.

3.2 Small Example: Swap Two Elements

Let us show a small example — swap — of programming and verification with the array state monad. The action `swap(i, j)` takes i -th and j -th values of the first array by `astate_get`, and then set them reversely by `astate_set`.

Definition `swap (I : finType) {A : Type} {sig : Sign} (i j : I) :`
`AState ((I, A) :: sig) unit :=`
`x ← astate_get i; y ← astate_get j; astate_set i y;; astate_set j x.`

`x ← t1; t2 and t1;; t2` are “do”-like notations which are equivalent to `astate_bind t1 (fun x ⇒ t2)` and `astate_bind t1 (fun _ ⇒ t2)` respectively. Correctness of the swap action is described by the following lemma.

Lemma run_swap

```
(I : finType) (A : Type) (sig : Sign) (i j : I)
(f : {ffun I → A}) (fs : states_AState sig) :
run_AState (swap i j) (fs, f) = (fs, [ffun k ⇒ f (tperm i j k)], tt).
```

$tperm\ i\ j$: {perm I} is a permutation (bijection) on I which transposes i and j. $tperm\ i\ j\ k$: I is j if $k = i$, i if $k = i$ and otherwise k. This formulation is useful for reasoning on a sequence of `swap` actions, e.g., sorting algorithms.

Operations of the array state monad can be erased from the goal by the simplification tactic `rewrite /=`. More generally, the case analysis and the simplification work as the erasure.

```
...
=====
(fs,
ffun_set (fin_decode (fin_encode j)) (f (fin_decode (fin_encode i)))
(ffun_set (fin_decode (fin_encode i)) (f (fin_decode (fin_encode j))) f),
tt) = (fs, [ffun k ⇒ f ((tperm i j) k)], tt)
```

The encoding/decoding functions can be erased by the `fin_encodeK` lemma. Both sides of equation have the form of `(fs, _, tt)`, thus we use the congruence rule.

```
congr (_, _, _); rewrite !fin_encodeK.
```

```
...
=====
ffun_set j (f i) (ffun_set i (f j) f) = [ffun k ⇒ f ((tperm i j) k)]
```

To prove a equation of finfuns, we use the lemma of functional extensionality `ffunP`. Applications of finfuns can be unfolded by the `ffunE` lemma.

```
apply/ffunP ⇒ k; rewrite !ffunE /=.
```

```
...
k : I
=====
(if k == j then f i else if k == i then f j else f k) = f ((tperm i j) k)
```

The remaining goal can be proved by case analysis on comparison and `tperm`.

```
case: tpermP; do!case: eqP; congruence. Qed.
```

The lift operator is also erased by similar method. Here is a correctness proof of the lifted swap action.

Global Opaque swap.

Lemma run_lift_swap

```
(I I' : finType) (A B : Type) (sig : Sign) (i j : I)
(f : {ffun I → A}) (g : {ffun I' → B}) (fs : states_AState sig) :
run_AState (sig := [:: (I', B), (I, A) & sig])
(ystate_lift (swap i j)) (fs, f, g) =
(fs, [ffun k ⇒ f (tperm i j k)], g, tt).
```

```
Proof. by rewrite /= run_swap. Qed.
```

4 Optimizations by an Improved Extraction Plugin

We provide two modifications for the extraction plugin to improve the efficiency of extracted programs, particularly which use the array state monad. The Coq program extraction translates Gallina² programs to target languages (OCaml, Haskell, Scheme, and JSON) and consists of three translations: 1. extraction from Gallina to MiniML, an intermediate abstract language for program extraction, 2. simplification (optimization) of MiniML terms, and 3. translation from MiniML to target languages.

The modifications are of the part 2 and 1 of the translation. The former reduces the construction and destruction costs of (co)inductive objects by inlining. The latter reduces the function call costs by applying η -expansion to match expressions and distributing the added arguments to each branch. Each optimization is particularly effective for programs using the MathComp library and monadic programs respectively.

4.1 Destructing Large Records

The MathComp library provides many mathematical structures [4] as canonical structures, e.g., `eqType`, `choiceType`, `countType`, `finType`, etc. which are represented as nested records in extracted programs. For example, the modified `finType` definition is translated to a nested record with 5 constructors and 11 fields by the program extraction. We implemented additional simplification mechanisms for MiniML terms to prevent performance degradation caused by handling such large records.

This section describes simplification rules for MiniML terms provided by the original and improved extraction plugin. The rules act on type coercion (`Obj.magic` in OCaml, and `unsafeCoerce` in Haskell) are omitted here because of simplicity. The key simplification rule for unfolding pattern matchings provided by the original extraction plugin is generalized ι -reduction relation.

Definition 1 (Generalized ι -reduction). *We inductively define an auxiliary relation $\rightsquigarrow_{\iota}^{\overline{cl}}$ for a sequence of pattern-matching clauses $\overline{cl} = C_1(\overline{x}_1) \rightarrow u_1 \mid \cdots \mid C_n(\overline{x}_n) \rightarrow u_n$ by the following three rules:*

$$C_i(t_1, \dots, t_m) \rightsquigarrow_{\iota}^{\overline{cl}} \begin{array}{l} \text{let } x_{i,1} := t_1 \text{ in } \dots \\ \text{let } x_{i,m} := t_m \text{ in } u_i \end{array} \quad (1)$$

$$t \rightsquigarrow_{\iota}^{\overline{cl}} t' \Rightarrow \text{let } x := u \text{ in } t \rightsquigarrow_{\iota}^{\overline{cl}} \text{let } x := u \text{ in } t' \quad (2)$$

$$t_1 \rightsquigarrow_{\iota}^{\overline{cl}} t'_1 \wedge \cdots \wedge t_m \rightsquigarrow_{\iota}^{\overline{cl}} t'_m \Rightarrow \begin{array}{ccc} \text{match } t \text{ with} & & \text{match } t \text{ with} \\ \left| \begin{array}{l} D_1(\overline{x}_1) \rightarrow t_1 \\ \dots \\ D_m(\overline{x}_m) \rightarrow t_m \end{array} \right. & \rightsquigarrow_{\iota}^{\overline{cl}} & \left| \begin{array}{l} D_1(\overline{x}_1) \rightarrow t'_1 \\ \dots \\ D_m(\overline{x}_m) \rightarrow t'_m \end{array} \right. \end{array} \quad (3)$$

The generalized ι -reduction relation \rightsquigarrow_{ι} is defined as follows:

$$t \rightsquigarrow_{\iota}^{\overline{cl}} t' \Rightarrow (\text{match } t \text{ with } \overline{cl}) \rightsquigarrow_{\iota} t' \quad (4)$$

² The specification language of Coq.

The combination of the rules (1) and (4) provides simple ι -reduction. The rules (2) and (3) allows to traverse nested match and let expressions in the head of term t in the rule (4).

Other simplification rules are listed below.

$$(\lambda x. t) u \rightsquigarrow \text{let } x := u \text{ in } t \quad (5)$$

$$\text{let } x := u \text{ in } t \rightsquigarrow t[x := u] \quad (t \text{ or } u \text{ is atomic,} \\ \text{or } x \text{ occurs at most once)} \quad (6)$$

$$(\text{let } x := t_1 \text{ in } t_2) u_1 \dots u_n \rightsquigarrow \text{let } x := t_1 \text{ in } t_2 u_1 \dots u_n \quad (7)$$

$$\begin{array}{ccc} \text{(match } t \text{ with} & & \text{match } t \text{ with} \\ | C_1(\bar{x}_1) \rightarrow t_1 & \rightsquigarrow & | C_1(\bar{x}_1) \rightarrow t_1 u_1 \dots u_m \\ | \dots & & | \dots \\ | C_n(\bar{x}_n) \rightarrow t_n & & | C_n(\bar{x}_n) \rightarrow t_n u_1 \dots u_m \\ \text{)} u_1 \dots u_m & & \end{array} \quad (8)$$

$$\begin{array}{ccc} \text{match } t \text{ with} & & \lambda y_1 \dots y_m. \text{match } t \text{ with} \\ | C_1(\bar{x}_1) \rightarrow \lambda y_1 \dots y_m. t_1 & \rightsquigarrow & | C_1(\bar{x}_1) \rightarrow t_1 \\ | \dots & & | \dots \\ | C_n(\bar{x}_n) \rightarrow \lambda y_1 \dots y_m. t_n & & | C_n(\bar{x}_n) \rightarrow t_n \end{array} \quad (9)$$

$$\text{let } x := (\text{let } y := t_1 \text{ in } t_2) \text{ in } t_3 \rightsquigarrow \text{let } y := t_1 \text{ in } (\text{let } x := t_2 \text{ in } t_3) \quad (10)$$

$$\text{let } x := C(t_1, \dots, t_n) \text{ in } u \rightsquigarrow \begin{array}{l} \text{let } y_1 := t_1 \text{ in } \dots \text{let } y_n := t_n \text{ in} \\ \text{let } x := C(y_1, \dots, y_n) \text{ in } u' \end{array} \quad (11)$$

where u' in the rule (11) is obtained by replacing all the subterms of the form $(\text{match } x \text{ with } \dots | C(z_1, \dots, z_n) \rightarrow t | \dots)$ in the u with the term $t[z_1 := y_1, \dots, z_n := y_n]$ which is a ι -reduced term of $(\text{match } C(y_1, \dots, y_n) \text{ with } \dots | C(z_1, \dots, z_n) \rightarrow t | \dots)$.

All simplification rules are safe for purely functional programs, but not safe for OCaml programs generally. The rules (6), (8) and (9) may change the execution order of code and skip executing some code, and other rules also help to apply these rules. Therefore it is difficult to implement these rules as optimizations for OCaml programs, and it is appropriate to implement it as a MiniML optimizer.

The generalized ι -reduction without the rule (2) and the rules (5) through (9) are provided by the original extraction plugin. We additionally implemented the rules (2), (10), and (11) to it. The rule (11) recursively destructs nested records, and the rule (10) assist it in the case of a term of the form $(\text{let } x := (\text{let } y := t \text{ in } C(t_1, \dots, t_n)) \text{ in } \dots)$ occurred.

Let us exemplify a simplification process of the following definition³:

```
Definition example : nat → nat → nat → bool :=
  let T := nat_eqType in fun x y z : T => (x == y) || (x == z) || (y == z).
```

³ `nat_eqType` is the canonical `eqType` instance of `nat`.

The simplification process is as follows. Constants inlined by extraction are underlined here. Identifiers `nat_eqType`, `Equality.Pack`, and `Equality.Mixin` are abbreviated here as `nateq`, `Packeq`, and `Mixineq`.

$$\begin{aligned}
 & \text{let } T := \underline{\text{nat}_{\text{eq}}} \text{ in} \\
 & \lambda x y z. \underline{\text{eq_op}} T x y \vee \underline{\text{eq_op}} T x z \vee \underline{\text{eq_op}} T y z \\
 = & \text{let } T := \text{Pack}_{\text{eq}} (\text{Mixin}_{\text{eq}} \text{eqn } \dots) \text{ in } \lambda x y z. & \text{(unfold)} \\
 & (\lambda T. \text{match } T \text{ with Pack}_{\text{eq}} (\text{Mixin}_{\text{eq}} f _) \rightarrow f) T x y \\
 & \vee (\lambda T. \text{match } T \text{ with Pack}_{\text{eq}} (\text{Mixin}_{\text{eq}} f _) \rightarrow f) T x z \\
 & \vee (\lambda T. \text{match } T \text{ with Pack}_{\text{eq}} (\text{Mixin}_{\text{eq}} f _) \rightarrow f) T y z \\
 \rightsquigarrow^* & \text{let } T := \text{Pack}_{\text{eq}} (\text{Mixin}_{\text{eq}} \text{eqn } \dots) \text{ in} & \text{(rules (5)} \\
 & \lambda x y z. (\text{match } T \text{ with Pack}_{\text{eq}} (\text{Mixin}_{\text{eq}} f _) \rightarrow f) x y & \text{and (6))} \\
 & \vee (\text{match } T \text{ with Pack}_{\text{eq}} (\text{Mixin}_{\text{eq}} f _) \rightarrow f) x z \\
 & \vee (\text{match } T \text{ with Pack}_{\text{eq}} (\text{Mixin}_{\text{eq}} f _) \rightarrow f) y z \\
 \rightsquigarrow & \text{let } a := \text{Mixin}_{\text{eq}} \text{eqn } \dots \text{ in let } T := \text{Pack}_{\text{eq}} a \text{ in} & \text{(rule (11))} \\
 & \lambda x y z. (\text{match } a \text{ with Mixin}_{\text{eq}} f _ \rightarrow f) x y \\
 & \vee (\text{match } a \text{ with Mixin}_{\text{eq}} f _ \rightarrow f) x z \\
 & \vee (\text{match } a \text{ with Mixin}_{\text{eq}} f _ \rightarrow f) y z \\
 \rightsquigarrow & \text{let } c := \text{eqn} \text{ in let } b := \dots \text{ in let } a := \text{Mixin}_{\text{eq}} c b \text{ in} & \text{(rule (11))} \\
 & \text{let } T := \text{Pack}_{\text{eq}} a \text{ in } \lambda x y z. c x y \vee c x z \vee c y z \\
 \rightsquigarrow^* & \lambda x y z. \text{eqn } x y \vee \text{eqn } x z \vee \text{eqn } y z & \text{(rule (6))}
 \end{aligned}$$

T can be unfolded by rules (6) and (4) if T has occurred only once. However, we need the rule (11) to apply the ι -reduction over the `let` expressions which cannot be simplified by the rule (6).

4.2 η -expansion on Case Analysis

Match expressions returning a function are commonly used in monadic programming and dependently typed programming. However, they increase the number of function calls and closure allocations and decrease the performance of programs. Let us consider the following monadic program that branches depending on whether the integer state is even or odd:

$$\text{get} \gg\gg \lambda n : \mathbb{Z}. \text{if } n \bmod 2 = 0 \text{ then } f \text{ else } g.$$

By unfolding the `get` and `\gg\gg` in the above program, a match expression returning a function⁴ can be found:

$$\lambda n : \mathbb{Z}. (\text{if } n \bmod 2 = 0 \text{ then } f \text{ else } g) n.$$

Another example from dependently typed programming is a map function for size-fixed vectors:

⁴ `if` expressions are syntax sugar for match expressions in Coq.

```
Fixpoint vec (n : nat) (A : Type) : Type :=
  if n is S n' then (A * vec n' A) else unit.
```

```
Fixpoint vmap (A B : Type) (f : A → B) (n : nat) : vec n A → vec n B :=
  if n is S n' then fun '(h, t) => (f h, vmap f t) else fun _ => tt.
```

The match expressions in the above examples can be optimized by the rules (8) and (9) respectively, and those rules can be applied for many other cases. However, these rules are not complete because of its syntactic restriction: match expressions to which that rules are applied should have following arguments or have λ -abstractions for each branch. Therefore, we apply the full η -expansion on all match expressions in the process of extraction from Gallina to MiniML.⁵ The rule (8) can be applied for η -expanded match expressions.

We additionally provide new Vernacular command `Extract Type Arity` to declare the arity of an inductive type which is realized by the `Extract Inductive` command, because the Coq system cannot recognize that the arity of `AState` is 1. Declared arities are used for full η -expansion described above. This command can be used as follows: `Extract Type Arity AState 1`.

5 Case Studies

This section demonstrates our library and improved extraction plugin using two applications: the union–find data structure and the quicksort algorithm. Here we provide an overview of formalizations and the performance comparison between the extracted code and other implementations for each application.

All the benchmark programs were compiled by OCaml 4.05.0+flambda with optimization flags `-O3 -remove-unused-arguments -unbox-closures` and performed on a Intel Core i5-7260U CPU @ 2.20 GHz equipped with 32 GB of RAM. Full major garbage collection and heap compaction are invoked before each measurement. All benchmark results represent the median of 5 different measurements with same parameters.

5.1 The Union–find Data Structure

We implemented and verified the union–find data structure with the path compression and the weighted union rule [18, 19]. The formalization takes 586 lines, and most key properties and reasoning on the union–find can be easily written out by using the `path` and `fingraph` library.

The benchmark results are shown in the Fig. 1. Here we compare the execution times of the OCaml code extracted from above formalization (optimized and unoptimized⁶ version) and handwritten OCaml implementation of same algorithm. The procedure to be measured here is the sequence of union n times

⁵ Implementing it as a part of the simplification of MiniML terms is difficult, because MiniML is a type-free language.

⁶ It is extracted by disabling new optimization mechanisms described in Sect. 4, but compiled with same OCaml compiler and optimization flags.

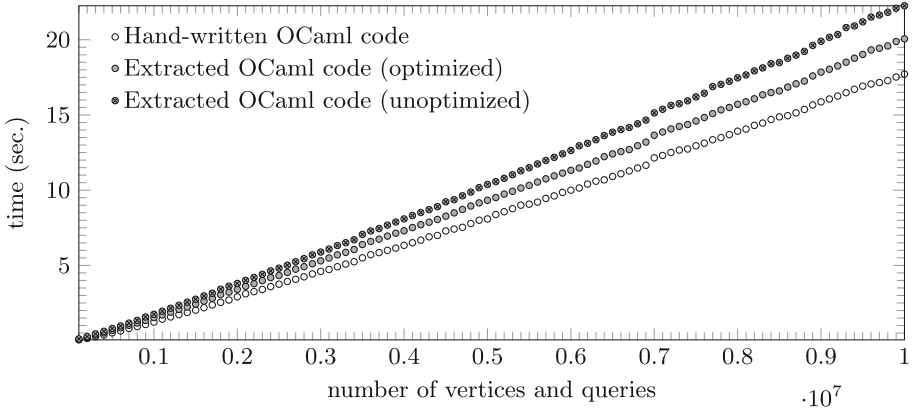


Fig. 1. Benchmark results of the union–find data structure

and find n times on n vertices union–find data structure, where all parameters of unions and finds are randomly selected. The time complexity of this procedure is $\Theta(n\alpha(n, n))$ where α is a functional inverse of Ackermann’s function. The results indicate that the OCaml implementation is about 1.1 times faster than the optimized Coq implementation, the optimized Coq implementation is about 1.1 times faster than the unoptimized Coq implementation, and the execution times of all implementations increase slightly more than linear.

5.2 The Quicksort Algorithm

We implemented and verified the quicksort algorithm by using the array state monad. The formalization including partitioning and the upward/downward search takes 365 lines, and the key properties are elegantly written out by using the theory of permutations. Here we explain the formalization techniques used for the quicksort algorithm by taking the partitioning function as an example. The partitioning function for I indexed arrays of A and comparison function $\text{cmp} : A \rightarrow A \rightarrow \text{bool}$ ⁷ has the following type:

`partition : A → ∀i j : 'I_#|I|. +1, i ≤ j → AState [:: (I, A)] 'I_#|I|. +1`

The `partition` takes a pivot of type and range of partition represented by indices $i\ j : 'I_#|I|. +1$, reorders the elements of the arrays from index i to $j - 1$ so that elements less than the pivot come before all the other elements, and returns the partition position. We proved the correctness of `partition` as the following lemma:

⁷ Here we assume that `cmp` is a total order and means “less than or equal to” in some sense.

```

CoInductive partition_spec
  (pivot : A) (i j : 'I_#|I|. +1) (arr : {ffun I → A}) :
  unit * {ffun I → A} * 'I_#|I|. +1 → Prop :=
  PartitionSpec (p : {perm I}) (k : 'I_#|I|. +1) :
  let arr' := [ffun ix ⇒ arr (p ix)] in
  (* 1 *) i ≤ k ≤ j →
  (* 2 *) perm_on [set ix | i ≤ fin_encode ix < j] p →
  (* 3 *) (∀ix : 'I_#|I|, i ≤ ix < k → ¬ cmp pivot (arr' (fin_decode ix))) →
  (* 4 *) (∀ix : 'I_#|I|, k ≤ ix < j → cmp pivot (arr' (fin_decode ix))) →
  partition_spec pivot i j arr (tt, arr', k).
  
```

```

Lemma run_partition
  (pivot : A) (i j : 'I_#|I|. +1) (Hij : i ≤ j) (arr : {ffun I → A}) :
  partition_spec pivot i j arr (run_AState (partition pivot Hij) (tt, arr)).
  
```

The `run_partition` can be applied for goals including some executions of `partition` without giving concrete parameters by using the simple idiom `case: run_partition`, and it obtains the properties of `partition` described below. It is achieved by a `Ssreflect` convention [11, Sect. 4.2.1], which means separating the specification from the lemma as the coinductive type family `partition_spec`.

In the specification `partition_spec`, the finfun `arr`, permutation `p`, and index `k` indicates the initial state, permutation performed by the partition, and partition position respectively, and the final state `arr'` is represented by

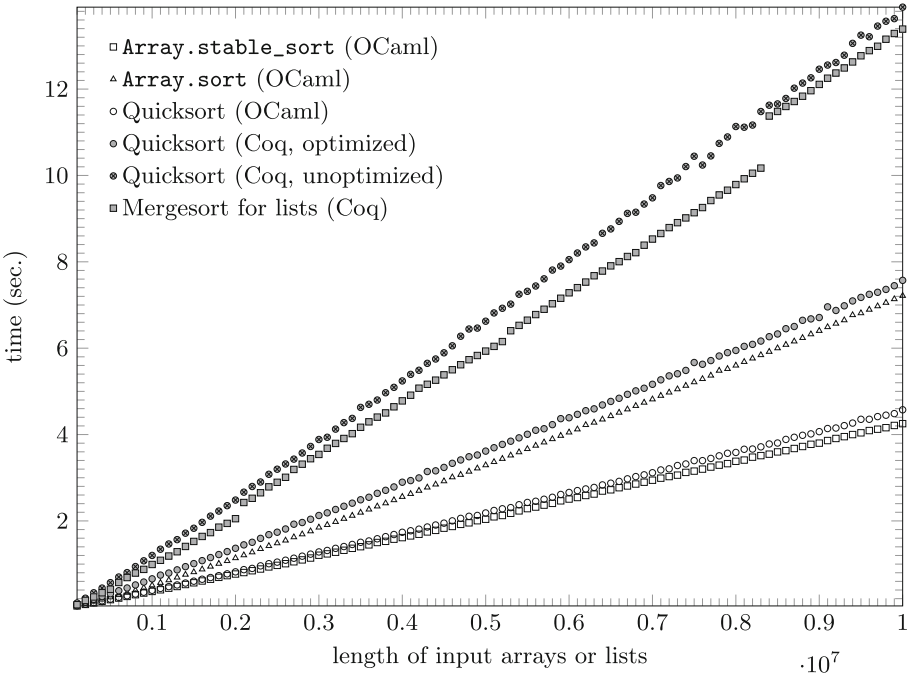


Fig. 2. Benchmark results of the quicksort and mergesort

`[ffun ix \Rightarrow arr (p ix)]` which means the finfun `arr` permuted by `p`. Properties of the partition are given as arguments of `PartitionSpec` and numbered from 1 to 4 in above code. Each property has the following meaning:

1. the partition position `k` is between `i` and `j`,
2. the partition only replaces value in the range from `i` to `j - 1`,
3. values in the range from `i` to `k - 1` are less than the pivot, and
4. values in the range from `k` to `j - 1` are greater than or equal to the pivot.

Of particular interest is that the property 2 can be written in the form of `perm_on A p` which means that the permutation `p` only displaces elements of `A`. `perm_on` is used for constructing algebraic theory in the `MathComp` library, but is also helpful for reasoning on sorting algorithms.

The benchmark results are shown in the Fig. 2. Here we compare the execution times of following implementations of sorting algorithms for randomly generated arrays or lists of integers: 1. `Array.stable_sort` and 2. `Array.sort` taken from OCaml standard library, 3. handwritten OCaml implementation of the quicksort, 4. optimized and 5. unoptimized OCaml code extracted from above formalization, and 6. OCaml code extracted from a Coq implementation of the bottom-up merge-sort algorithm for lists. The results indicate that the implementation 5 (quicksort in Coq, unoptimized) is slowest of those, the implementation 4 (quicksort in Coq, optimized) is 1.7–1.8 times faster than implementations 5 and 6, and OCaml implementations 1, 2 and 3 are 1.07–1.8 times faster than the implementation 3.

6 Related Work

`Ynot` [13] is a representative Coq library for verification and extraction of higher-order imperative programs. `Ynot` supports various kind of side-effects, separation-logic-based reasoning method, and automation facility [3] for it, and provides many example implementations of and formal proofs for data structures and algorithms including the union–find data structure. The formal development of the union–find provided by `Ynot` takes 1,067 lines of code, and our formal development of it (see Sect. 5.1) takes 586 lines. Both implementations use almost the same optimization strategies: the union by rank and the weighted union rule respectively, and the path compression. This comparison indicates that `Ynot` is good at its expressiveness, but our method has smaller proof burden.

7 Conclusion

We have established a novel, lightweight, and axiom-free method for verification and extraction of efficient effectful programs using mutable arrays in Coq. This method consists of the following two parts: a state monad specialized for mutable array programming (the array state monad) and an improved extraction plugin for Coq. The former enables a simple reasoning method for and safe extraction of efficient effectful programs. The latter optimizes programs extracted from formal developments using our library, and it is also effective for mathematical structures provided by the `MathComp` library and monadic programs.

We would like to improve this study with more expressive array state monad, large and realistic examples, and correctness proof of our extraction method for effectful programs.

Acknowledgments. We thank Yuki Yoshi Kameyama and anonymous referees for valuable comments on an earlier version of this paper. This work was supported by JSPS KAKENHI Grant Number 17J01683.

References

1. Brady, E.: Programming and reasoning with algebraic effects and dependent types. In: ICFP 2013, pp. 133–144. ACM (2013)
2. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_14
3. Chlipala, A., Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Effective interactive proofs for higher-order imperative programs. In: ICFP 2009, pp. 79–90. ACM (2009)
4. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 327–342. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_23
5. Launchbury, J., Peyton Jones, S.L.: Lazy functional state threads. In: PLDI 1994, pp. 24–35. ACM (1994)
6. Leroy, X.: A formally verified compiler back-end. *J. Autom. Reason.* **43**(4), 363–446 (2009)
7. Letouzey, P.: A new extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-39185-1_12
8. Letouzey, P.: Programmation fonctionnelle certifiée - L'extraction de programmes dans l'assistant Coq. Ph.D. thesis, Université Paris-Sud (2004)
9. Letouzey, P.: Extraction in Coq: an overview. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) CiE 2008. LNCS, vol. 5028, pp. 359–369. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69407-6_39
10. Mahboubi, A., Tassi, E.: Canonical structures for the working Coq user. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 19–34. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_5
11. Mahboubi, A., Tassi, E.: Mathematical components (2016). <https://math-comp.github.io/mcb/book.pdf>
12. Nanevski, A., Morrisett, G., Birkedal, L.: Hoare type theory, polymorphism and separation. *J. Funct. Prog* **18**(5–6), 865–911 (2008)
13. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: dependent types for imperative programs. In: ICFP 2008, pp. 229–240. ACM (2008)
14. O’Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44802-0_1
15. Paulin-Mohring, C.: Extracting F_ω ’s programs from proofs in the Calculus of Constructions. In: POPL 1989, pp. 89–104. ACM (1989)

16. Sakaguchi, K., Kameyama, Y.: Efficient finite-domain function library for the Coq proof assistant. *IPSPJ Trans. Prog.* **10**(1), 14–28 (2017)
17. Swamy, N., Hrițcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: *POPL 2016*, pp. 256–270. ACM (2016)
18. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. *J. ACM* **22**(2), 215–225 (1975)
19. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. *J. ACM* **31**(2), 245–281 (1984)
20. The Coq Development Team: The Coq Proof Assistant Reference Manual (2017). <https://coq.inria.fr/distrib/V8.7.0/refman/>
21. The Mathematical Components Project: The mathematical components repository. <https://github.com/math-comp/math-comp>
22. Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) *AFP 1995. LNCS*, vol. 925, pp. 24–52. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59451-5_2



Functional Pearl: Folding Polynomials of Polynomials

Chen-Mou Cheng^{1(✉)}, Ruey-Lin Hsu², and Shin-Cheng Mu³

¹ National Taiwan University, Taipei, Taiwan
ccheng@cc.ee.ntu.edu.tw

² National Central University, Taoyuan, Taiwan
petercommand@gmail.com

³ Academia Sinica, Taipei, Taiwan
scm@iis.sinica.edu.tw

Abstract. Polynomials are a central concept to many branches in mathematics and computer science. In particular, manipulation of polynomial expressions can be used to model a wide variety of computation. In this paper, we consider a simple recursive construction of multivariate polynomials over a base ring such as the integers or a (finite) field. We show that this construction allows inductive implementation of polynomial operations such as arithmetic, evaluation, substitution, etc. Furthermore, we can transform a polynomial expression into a sequence of arithmetic expressions in the base ring and prove the correctness of this transformation in Agda. Combined with our recursive construction, this allows for compiling polynomial expressions over a tower of extension fields into scalar expressions over the ground field, for example. Such a technique is not only interesting in its own right but also finds plentiful application in research areas such as cryptography.

1 Introduction

A *univariate polynomial* over a base ring R is a finite sum of the form

$$a_n X^n + a_{n-1} X^{n-1} + \cdots + a_0,$$

where $a_i \in R$ are the coefficients, and X is called an *indeterminate*. The set of univariate polynomials over R forms a ring, denoted as $R[X]$. We can allow two or more indeterminates X_1, X_2, \dots, X_m and thus arrive at a *multivariate polynomial*, a finite sum of the form

$$\sum_i a_i X_1^{e_1^{(i)}} X_2^{e_2^{(i)}} \cdots X_m^{e_m^{(i)}},$$

where $a_i \in R$ are the coefficients, and the nonnegative integers $e_j^{(i)}$ are the exponents. The set of m -variate polynomials over R , denoted as $R[X_1, X_2, \dots, X_m]$, also forms a ring, referred to as a *ring of polynomials*.

Polynomials are a central concept to many branches in mathematics and computer science. In particular, manipulation of polynomial expressions can be used to model a wide variety of computation. For example, every element of an algebraic extension field F over a base field K can be identified as a polynomial over K , e.g., cf. Theorem 1.6, Chap. 5 of the (standard) textbook by Hungerford [6]. Addition in F is simply polynomial addition over K , whereas multiplication in F is polynomial multiplication modulo the defining polynomial of F over K . Let us use the familiar case of the complex numbers over the real numbers as a concrete example. The complex numbers can be obtained by adjoining to the real numbers a root i of the polynomial $X^2 + 1$. In this case, every complex number can be identified as a polynomial $a + bi$ for a, b real. The addition of $a_1 + b_1i$ and $a_2 + b_2i$ is simply $(a_1 + a_2) + (b_1 + b_2)i$, whereas the multiplication, $(a_1 + b_1i)(a_2 + b_2i) \bmod i^2 + 1 = (a_1a_2 - b_1b_2) + (a_1b_2 + a_2b_1)i$.

In addition to arithmetic in an algebraic extension field, manipulation of polynomial expressions also finds rich application in cryptography in particular. A wide variety of cryptosystems can be implemented using polynomial expressions over a finite field or $\mathbb{Z}/n\mathbb{Z}$, the ring of integers modulo n . In elliptic curve cryptography [8], for example, we use the group structure of certain elliptic curves over finite fields to do cryptography, and the group laws are often given in polynomial expressions over finite fields. Another example is certain classes of post-quantum cryptosystems, i.e., those expected to survive large quantum computers' attack. Among the most promising candidates, we have multivariate cryptosystems [3] and several particularly efficient lattice-based cryptosystems [4, 7], for which encryption and decryption operations can be carried out using polynomial expressions over finite fields or $\mathbb{Z}/n\mathbb{Z}$.

This pearl is initially motivated by our research in cryptography, where we often have to deal with multivariate polynomials over various base rings, as exemplified above. We also need to transform a polynomial expression into a sequence of arithmetic expressions over its base ring. This is useful for, e.g., software implementation of cryptosystems on microprocessors with no native hardware support for arithmetic operations with polynomials or integers of cryptographic sizes, which typically range from a few hundreds to a few thousands of bits. Again using multiplication of two complex numbers as an example, we would need a sequence of real arithmetic expressions for implementing $z = z_r + iz_i = (x_r + ix_i) \times (y_r + iy_i) = xy$:

$$\begin{aligned} t_1 &\leftarrow x_r \times y_r; \\ t_2 &\leftarrow x_i \times y_i; \\ t_3 &\leftarrow x_r \times y_i; \\ t_4 &\leftarrow x_i \times y_r; \\ z_r &\leftarrow t_1 - t_2; \\ z_i &\leftarrow t_3 + t_4. \end{aligned}$$

Furthermore, we would like to have a precision that exceeds what our hardware can natively support. For example, let us assume that we have a machine with

native support for an integer type $-R < x < R$. In this case, we split each variable ζ into a low part plus a high part: $\zeta = \zeta^{(0)} + R\zeta^{(1)}$, $-R < \zeta^{(0)}, \zeta^{(1)} < R$. Now let us assume that our machine has a multiplication instruction $(c^{(0)}, c^{(1)}) \leftarrow a \times b$ such that $-R < a, b, c^{(0)}, c^{(1)} < R$ and $ab = c^{(0)} + Rc^{(1)}$. For simplicity, let us further assume that our machine has n -ary addition instructions for $n = 2, 3, 4$: $(c^{(0)}, c^{(1)}) \leftarrow a_1 + \dots + a_n$ such that $-R < a_1, \dots, a_n, c^{(0)}, c^{(1)} < R$ and $a_1 + \dots + a_n = c^{(0)} + Rc^{(1)}$. We can then have a suboptimal yet straightforward implementation of, say, $t_1 = t_1^{(0)} + Rt_1^{(1)} + R^2t_1^{(2)} + R^3t_1^{(3)} = (x_r^{(0)} + Rx_r^{(1)}) \times (y_r^{(0)} + Ry_r^{(1)}) = x_r \times y_r$ as follows.

$$\begin{array}{ll}
(t_1^{(0)}, s_1^{(1)}) \leftarrow x_r^{(0)} \times y_r^{(0)}; & //t_1^{(0)} + Rs_1^{(1)} \\
(s_2^{(0)}, s_2^{(1)}) \leftarrow x_r^{(0)} \times y_r^{(1)}; & //Rs_2^{(0)} + R^2s_2^{(1)} \\
(s_3^{(0)}, s_3^{(1)}) \leftarrow x_r^{(1)} \times y_r^{(0)}; & //Rs_3^{(0)} + R^2s_3^{(1)} \\
(s_4^{(0)}, s_4^{(1)}) \leftarrow x_r^{(1)} \times y_r^{(1)}; & //R^2s_4^{(0)} + R^3s_4^{(1)} \\
(t_1^{(1)}, s_5^{(1)}) \leftarrow s_1^{(1)} + s_2^{(0)} + s_3^{(0)}; & //Rt_1^{(1)} + R^2s_5^{(1)} \\
(t_1^{(2)}, s_6^{(1)}) \leftarrow s_2^{(1)} + s_3^{(1)} + s_4^{(0)} + s_5^{(1)}; & //R^2t_1^{(2)} + R^3s_6^{(1)} \\
(t_1^{(3)}, _) \leftarrow s_4^{(1)} + s_6^{(1)}. & //R^3t_1^{(3)}
\end{array}$$

It might be surprising that, with the advance of compiler technology today, such programs are still mostly coded and optimized manually, sometimes in assembly language, for maximal efficiency. Naturally, we would like to automate this process as much as possible. Furthermore, with such security-critical applications, we would like to have machine-verified proofs that the transformation and optimizations are correct.

In attempting toward this goal, we have come up with this pearl. It is not yet practical but, we think, is neat and worth documenting. A key observation is that there is an isomorphism between multivariate polynomial ring $R[X_1, X_2, \dots, X_m]$ and univariate polynomial ring $S[X_m]$ over the base ring $S = R[X_1, X_2, \dots, X_{m-1}]$, cf. Corollary 5.7, Chap.3 of Hungerford [6]. This allows us to define a datatype `Poly` for univariate polynomials, and reuse it inductively to define multivariate polynomials — an n -variate polynomial can be represented by `Polyn` (`Poly` applied n times). Most operations on the polynomials can be defined either in terms of the *fold* induced by `Poly`, or by induction on n , hence the title.

We explore the use of `Polyn` and its various implications in depth in Sect. 2. Then in Sect. 3, we present implementations of common polynomial operations such as evaluation, substitution, etc. We pay special attention to an operation `expand` and prove its correctness, since it is essential in transforming polynomial into scalar expressions. In Sect. 4, we show how to compile a polynomial function into an assembly program that computes it. We present a simple compilation, also defined in terms of `fold`, and prove its correctness, while leaving further optimization to future work. The formulation in this pearl have been implemented in both Haskell and Agda [9], the latter also used to verify our proofs. The code is available on line at <https://github.com/petercommand/ExtFieldComp>.

2 Univariate and Multivariate Polynomials

In this section, we present our representation for univariate and multivariate polynomials, as well as their semantics. The following Agda datatype denotes a univariate polynomial whose coefficients are of type A :¹

```
data Poly (A : Set) : Set where
  Ind  : Poly A
  Lit  : A → Poly A
  (:+) : Poly A → Poly A → Poly A
  (:×) : Poly A → Poly A → Poly A ,
```

where `Ind` denotes the indeterminate, `Lit` denotes a constant (of type A), while `(:+)` and `(:×)` respectively denote addition and multiplication. A polynomial $2x^2 + 3x + 1$ can be represented by the following expression of type `Poly ℤ`:

```
(Lit 2 :× Ind :× Ind) :+ (Lit 3 :× Ind) :+ Lit 1 .
```

Notice that the type parameter A is abstracted over the type of coefficients. This allows us to represent polynomials whose coefficients have non-simple types — in particular, polynomials whose coefficients are themselves polynomials. Do not confuse this with the more conventional representation of arithmetic expressions:

```
data Expr A = Var A | Lit Int | Expr A :+ Expr A | Expr A :× Expr A ,
```

where the literals are usually assigned a fixed type (in this example, `Int`), and the type parameter is abstracted over variables `Var`.

2.1 Univariate Polynomial and Its Semantics

In the categorical style outlined by Bird and de Moor [1], every *regular* datatype gives rise to a *fold*, also called a *catamorphism*. The type `Poly` induces a fold that, conventionally, takes four arguments, each replacing one of its four constructors. To facilitate our discussion later, we group the last two arguments together. The fold for `Poly` is thus given by:

```
foldP : {A B : Set} → B → (A → B) →
  ((B → B → B) × (B → B → B)) → Poly A → B
foldP x f ((⊕), (⊗)) Ind      = x
foldP x f ((⊕), (⊗)) (Lit y) = f y
foldP x f ((⊕), (⊗)) (e1 :+ e2) = foldP x f ((⊕), (⊗)) e1 ⊕
  foldP x f ((⊕), (⊗)) e2
foldP x f ((⊕), (⊗)) (e1 :× e2) = foldP x f ((⊕), (⊗)) e1 ⊗
  foldP x f ((⊕), (⊗)) e2 ,
```

where arguments `x`, `f`, `(⊕)`, and `(⊗)` respectively replace constructors `Ind`, `Lit`, `(:+)`, and `(:×)`.

¹ We use Haskell convention that infix data constructors start with a colon and, for concise typesetting, write `(:+)` instead of the Agda notation `_:+_`. In the rest of the paper we also occasionally use Haskell syntax for brevity.

Evaluation. To evaluate a polynomial of type $\text{Poly } A$, we have to know how to perform arithmetic operations for type A . Define

$$\begin{aligned} \text{Ring} &: \text{Set} \rightarrow \text{Set} \\ \text{Ring } A &= ((A \rightarrow A \rightarrow A) \times (A \rightarrow A \rightarrow A)) \times A \times A \times (A \rightarrow A) , \end{aligned}$$

the intention is that the tuple $\text{Ring } A$ defines addition, multiplication, zero, one, and negation for A (addition and multiplication are grouped together, for our convenience later). In our Haskell implementation, Ring is a type class for types whose addition and multiplication are defined. It can usually be inferred what instance of Ring to use. When proving properties about foldP , however, it is clearer to make the construction of Ring instances explicit.

With the presence of Ind , the semantics of $\text{Poly } A$ should be $A \rightarrow A \rightarrow A$ — a function that takes the value of the indeterminate and returns a value. We define the following operation that lifts pointwise the addition and multiplication of some type B to $A \rightarrow B$:

$$\begin{aligned} \text{ring}_{\rightarrow} &: \forall \{A B\} \rightarrow \text{Ring } B \rightarrow \text{Ring } (A \rightarrow B) \\ \text{ring}_{\rightarrow} &(((+), (\times)), \mathbf{0}, \mathbf{1}, \text{neg}) = \\ &((\lambda f g x \rightarrow f x + g x, \lambda f g x \rightarrow f x \times g x), \text{const } \mathbf{0}, \text{const } \mathbf{1}, (\text{neg} :)) , \end{aligned}$$

where $\text{const } x y = x$. The semantics of a univariate polynomial is thus given by:

$$\begin{aligned} \text{sem}_1 &: \forall \{A\} \rightarrow \text{Ring } A \rightarrow \text{Poly } A \rightarrow A \rightarrow A \\ \text{sem}_1 \text{ rng} &= \text{foldP id const (fst (ring}_{\rightarrow} \text{ rng}))} , \end{aligned}$$

where $\text{id } x = x$ and fst retrieves the left component of a pair.

2.2 Bivariate Polynomials

To represent polynomials with two indeterminates, one might extend Poly with a constructor Ind' in addition to Ind . It turns out to be unnecessary — it is known that the bivariate polynomial ring $R[X, Y]$ is isomorphic to $R[X][Y]$ (modulo the operation litDist , to be defined later). That is, a polynomial over base ring A with two indeterminates can be represented by $\text{Poly } (\text{Poly } A)$.

To understand the isomorphism, consider the following expression:

$$\begin{aligned} e &: \text{Poly } (\text{Poly } \mathbb{Z}) \\ e &= (\text{Lit } (\text{Lit } 3) : \times \text{Ind} : \times \text{Lit } (\text{Ind} : + \text{Lit } 4)) : + \text{Lit } \text{Ind} : + \text{Ind} . \end{aligned}$$

Note that to represent a literal 3, we have to write $\text{Lit } (\text{Lit } 3)$, since the first Lit takes a $\text{Poly } \mathbb{Z}$ as its argument. To evaluate e using sem_1 , we have to define $\text{Ring } (\text{Poly } \mathbb{Z})$. A natural choice is to connect two expressions using corresponding constructors:

$$\begin{aligned} \text{ringP} &: \forall \{A\} \rightarrow \text{Ring } A \rightarrow \text{Ring } (\text{Poly } A) \\ \text{ringP } (-, \mathbf{0}, \mathbf{1}, \text{neg}) &= (((:+), (:\times)), \text{Lit } \mathbf{0}, \text{Lit } \mathbf{1}, (\text{Lit } (\text{neg } \mathbf{1}) : \times)) . \end{aligned}$$

With ringP defined, $\text{sem}_1(\text{ringP } r) e$ has type $\text{Poly } A \rightarrow \text{Poly } A$. Evaluating, for example $\text{sem}_1(\text{ringP } r) e(\text{Ind } :+ \text{Lit } 1)$, yields

$$\begin{aligned} e' &: \text{Poly } \mathbb{Z} \\ e' &= (\text{Lit } 3 \times (\text{Ind } :+ \text{Lit } 1) \times (\text{Ind } :+ \text{Lit } 4)) :+ \text{Ind } :+ (\text{Ind } :+ \text{Lit } 1) . \end{aligned}$$

Note that $\text{Lit } \text{Ind}$ in e is replaced by the argument $\text{Ind } :+ \text{Lit } 1$. Furthermore, one layer of Lit is removed, thus both $\text{Lit } 3$ and $\text{Ind } :+ \text{Lit } 4$ are exposed to the outermost level. The expression e' may then be evaluated by $\text{sem}_1 \text{rng}\mathbb{Z}$, where $\text{rng}\mathbb{Z} : \text{Ring } \mathbb{Z}$. The result is a natural number. In general, the function sem_2 that evaluates $\text{Poly}(\text{Poly } A)$ can be defined by:

$$\begin{aligned} \text{sem}_2 &: \forall \{A\} \rightarrow \text{Ring } A \rightarrow \text{Poly}(\text{Poly } A) \rightarrow \text{Poly } A \rightarrow A \rightarrow A \\ \text{sem}_2 r e_2 e_1 \times &= \text{sem}_1 r (\text{sem}_1(\text{ringP } r) e_2 e_1) \times . \end{aligned}$$

This is how $\text{Poly}(\text{Poly } \mathbb{Z})$ simulates bivariate polynomials: the two indeterminates are respectively represented by Ind and $\text{Lit } \text{Ind}$. During evaluation, Ind can be instantiated to an expression arg of type $\text{Poly } \mathbb{Z}$, while $\text{Lit } \text{Ind}$ can be instantiated to a \mathbb{Z} . If arg contains Ind , it refers to the next indeterminate.

What about expressions like $\text{Lit}(\text{Ind } :+ \text{Lit } 4)$? One can see that its semantics is the same as $\text{Lit } \text{Ind } :+ \text{Lit}(\text{Lit } 4)$, the expression we get by pushing Lit to the leaves. In general, define the following function:

$$\begin{aligned} \text{litDist} &: \forall \{A\} \rightarrow \text{Poly}(\text{Poly } A) \rightarrow \text{Poly}(\text{Poly } A) \\ \text{litDist} &= \text{foldP } \text{Ind} (\text{foldP } (\text{Lit } \text{Ind}) (\text{Lit} \cdot \text{Lit}) ((:+) , (:\times))) ((:+) , (:\times)) . \end{aligned}$$

The function traverses through the given expression and, upon encountering a subtree $\text{Lit } e$, lifts e to $\text{Poly}(\text{Poly } A)$ while distributing Lit inwards e . We can prove the following theorem:

Theorem 1. *For all $e : \text{Poly}(\text{Poly } A)$ and $r : \text{Ring } A$, we have $\text{sem}_2 r (\text{litDist } e) = \text{sem}_2 r e$.*

2.3 Multivariate Polynomials

In general, as we have mentioned in Sect. 1, the multivariate polynomial $R[X_1, X_2, \dots, X_m]$ is isomorphic to univariate polynomial ring $S[X_m]$ over the base ring $S = R[X_1, X_2, \dots, X_{m-1}]$ (modulo the distribution law of Lit). That is, a polynomial over A with n indeterminates can be represented by $\text{Poly}^n A$, defined by

$$\begin{aligned} \text{Poly}^{\text{zero}} A &= A \\ \text{Poly}^{\text{suc } n} A &= \text{Poly}(\text{Poly}^n A) . \end{aligned}$$

To define the semantics of $\text{Poly}^n A$, recall that, among its n indeterminates, the outermost indeterminate shall be instantiated to an expression of

type $\text{Poly}^{n-1} A$, the next indeterminate to $\text{Poly}^{n-2} A$..., and the inner most indeterminate to A , before yielding a value of type A . Define

$$\begin{aligned} \text{DChain} &: \text{Set} \rightarrow \mathbb{N} \rightarrow \text{Set} \\ \text{DChain } A \text{ zero} &= \top \\ \text{DChain } A \text{ (suc } n) &= \text{Poly}^n A \times \text{DChain } A \ n \ , \end{aligned}$$

that is, $\text{DChain } A \ n$ (the name standing for a “descending chain”) is a list of n elements, with the first having type $\text{Poly}^{n-1} A$, the second $\text{Poly}^{n-2} A$, and so on. The type \top denotes the “unit” type, inhabited by exactly one term tt .

Given an implementation of $\text{Ring } A$, the semantics of $\text{Poly}^n A$ is a function $\text{DChain } A \ n \rightarrow A$, defined inductively as below:

$$\begin{aligned} \text{sem} : \forall \{A\} \rightarrow \text{Ring } A \rightarrow (n : \mathbb{N}) \rightarrow \text{Poly}^n A \rightarrow \text{DChain } A \ n \rightarrow A \\ \text{sem } r \text{ zero } \quad \times \text{tt} \quad &= x \\ \text{sem } r \text{ (suc } n) \text{ e (t , es)} &= \text{sem } r \ n \ (\text{sem}_1 (\text{ringP}^* r \ n) \text{ e t}) \text{ es} \ , \end{aligned}$$

where ringP^* delivers the $\text{Ring } (\text{Poly}^n A)$ instance for all n :

$$\begin{aligned} \text{ringP}^* : \forall \{A\} \rightarrow \text{Ring } A \rightarrow \forall n \rightarrow \text{Ring } (\text{Poly}^n A) \\ \text{ringP}^* r \text{ zero} \quad &= r \\ \text{ringP}^* r \text{ (suc } n) &= \text{ringP } (\text{ringP}^* r \ n) \ . \end{aligned}$$

For $n := 2$ and 3 , for example, $\text{sem } r \ n$ expands to:

$$\begin{aligned} \text{sem } r \ 2 \text{ e (t}_1, \text{t}_0, \text{tt)} &= \text{sem}_1 r \ (\text{sem}_1 (\text{ringP } r) \text{ e t}_1) \text{t}_0 \\ &= (\text{sem}_1 r \cdot \text{sem}_1 (\text{ringP } r) \text{ e}) \text{t}_1 \text{t}_0 \ , \\ \text{sem } r \ 3 \text{ e (t}_2, \text{t}_1, \text{t}_0, \text{tt)} &= \text{sem}_1 r \ (\text{sem}_1 (\text{ringP } r) \ (\text{sem}_1 (\text{ringP}^2 r) \text{ e t}_2) \text{t}_1) \text{t}_0 \\ &= (\text{sem}_1 r \cdot \text{sem}_1 (\text{ringP } r) \cdot \text{sem}_1 (\text{ringP}^2 r) \text{ e}) \text{t}_2 \text{t}_1 \text{t}_0 \ . \end{aligned}$$

Essentially, $\text{sem } r \ n$ is n -fold composition of $\text{sem}_1 (\text{ringP}^i r)$, each interpreting one level of the given expression.

3 Operations on Polynomials

Having defined a representation for multivariate polynomials, we ought to demonstrate that this representation is feasible — that we can define most of the operations we want. In fact, it turns that most of them can be defined either in terms of foldP or by induction over the number of iterations Poly is applied.

3.1 Rotation

The first operation swaps the two outermost indeterminates of a $\text{Poly}^2 A$, using foldP . This function witnesses the isomorphism between $R[X_1, \dots, X_{m-1}][X_m]$ and $R[X_m, X_1, \dots, X_{m-2}][X_{m-1}]$. It is instructive comparing it with litDist .

$$\begin{aligned} \text{rotaPoly}_2 : \forall \{A\} \rightarrow \text{Poly}^2 A \rightarrow \text{Poly}^2 A \\ \text{rotaPoly}_2 = \text{foldP } (\text{Lit } \text{Ind}) \ (\text{foldP } \text{Ind } (\text{Lit} \cdot \text{Lit}) \ ((:+) , (:\times))) \ ((:+) , (:\times)) \ . \end{aligned}$$

In rotaPoly_2 , the outermost Ind is replaced by Lit Ind . When encountering $\text{Lit } e$, the inner e is lifted to $\text{Poly}^2 A$. The Ind inside e remains Ind , which becomes the outermost indeterminate after lifting. Note that both litDist and rotaPoly_2 apply to $\text{Poly}^n A$ for all $n \geq 2$, since A can be instantiated to a polynomial as well.

Consider $\text{Poly}^3 A$, a polynomial with (at least) three indeterminates. To “rotate” the three indeterminates, that is, turn $\text{Lit}^2 \text{Ind}$ to Lit Ind , Lit Ind to Ind , and Ind to $\text{Lit}^2 \text{Ind}$, we can define:

$$\text{rotaPoly}_3 = \text{fmap } \text{rotaPoly}_2 \cdot \text{rotaPoly}_2 \text{ ,}$$

where fmap is the usual “functorial map” function for Poly :

$$\text{fmap} : \forall \{A B\} \rightarrow (A \rightarrow B) \rightarrow \text{Poly } A \rightarrow \text{Poly } B \text{ .}$$

The first rotaPoly_2 swaps the two outer indeterminates, while $\text{fmap } \text{rotaPoly}_2$ swaps the inner two. To rotate the outermost four indeterminates of a $\text{Poly}^4 A$, we may define:

$$\text{rotaPoly}_4 = \text{fmap } (\text{fmap } \text{rotaPoly}_2) \cdot \text{rotaPoly}_3 \text{ .}$$

In general, the following function rotates the first n indeterminates of the given polynomial:

$$\begin{aligned} \text{rotaPoly} : \forall \{A\} (n : \mathbb{N}) \rightarrow \text{Poly}^n A \rightarrow \text{Poly}^n A \\ \text{rotaPoly } \text{zero} &= \text{id} \\ \text{rotaPoly } (\text{suc } \text{zero}) &= \text{id} \\ \text{rotaPoly } (\text{suc } (\text{suc } \text{zero})) &= \text{rotaPoly}_2 \\ \text{rotaPoly } (\text{suc } (\text{suc } (\text{suc } n))) &= \text{fmap}^{\text{suc } n} \text{rotaPoly}_2 \cdot \text{rotaPoly } (\text{suc } (\text{suc } n)) \text{ .} \end{aligned}$$

Note that in the actual code we need to convince Agda that $\text{Poly}^n (\text{Poly } A)$ is the same type as $\text{Poly} (\text{Poly}^n A)$ and use subst to coerce between the two types. We omit those details for clarity.

Given m and n , $\text{rotaOuter } n m$ compose $\text{rotaPoly } n$ with itself m times. Therefore, the outermost n indeterminates are rotated m times. It will be handy in Sect. 3.2.

$$\begin{aligned} \text{rotaOuter} : \forall \{A\} (n m : \mathbb{N}) \rightarrow \text{Poly}^n A \rightarrow \text{Poly}^n A \\ \text{rotaOuter } n \text{ zero} &= \text{id} \\ \text{rotaOuter } n (\text{suc } m) &= \text{rotaOuter } n m \cdot \text{rotaPoly } n \text{ e} \text{ .} \end{aligned}$$

3.2 Substitution

Substitution is another operation that one would expect. Given an expression e , how do we substitute, for each occurrence of Ind , another expression e' , using operations we have defined? Noticing that the type of sem_1 can be instantiated to $\text{Poly}^2 A \rightarrow \text{Poly } A \rightarrow \text{Poly } A$, we may lift e to $\text{Poly}^2 A$ by wrapping it with Lit , do a rotaPoly_2 to swap the Ind in e to the outermost position, and use sem_1 to perform the substitution:

$$\begin{aligned} \text{substitute}_1 &: \forall \{A\} \rightarrow \text{Ring } A \rightarrow \text{Poly } A \rightarrow \text{Poly } A \rightarrow \text{Poly } A \\ \text{substitute}_1 \text{ r e e}' &= \text{sem}_1 (\text{ringP r}) (\text{rotaPoly}_2 (\text{Lit e})) \text{ e}' . \end{aligned}$$

What about $e : \text{Poly}^2 A$? We may lift it to $\text{Poly}^4 A$, perform two rotaPoly_4 to expose its two indeterminates, before using sem_2 :

$$\begin{aligned} \text{substitute}_2 &: \forall \{A\} \rightarrow \text{Ring } A \rightarrow \text{Poly}^2 A \rightarrow \text{Poly}^2 A \rightarrow \text{Poly}^2 A \rightarrow \text{Poly}^2 A \\ \text{substitute}_2 \text{ r e e}'' &= \\ &\text{sem}_2 (\text{ringP} (\text{ringP r})) (\text{rotaPoly}_4 (\text{rotaPoly}_4 \text{ Lit} (\text{Lit e}))) (\text{Lit e}') \text{ e}'' . \end{aligned}$$

Consider the general case with substituting the n indeterminates in $e : \text{Poly}^n A$ for n expressions, each of type $\text{Poly}^n A$. Let $\text{Vec } B \ n$ be the type of vectors (lists of fixed lengths) of length n . A general substitute can be defined by:

$$\begin{aligned} \text{substitute} &: \forall \{A\} \ n \rightarrow \text{Ring } A \rightarrow \text{Poly}^n A \rightarrow \text{Vec} (\text{Poly}^n A) \ n \rightarrow \text{Poly}^n A \\ \text{substitute } \{A\} \ n \text{ r e es} &= \\ &\text{sem} (\text{ringP}^* \text{ r } n) (\text{rotaOuter} (n + n) n (\text{liftPoly } n (n + n) e)) \\ &\quad (\text{toDChain es}) , \end{aligned}$$

where $\text{liftPoly } n \ m$ (with $n \leq m$) lifts a $\text{Poly}^n A$ to $\text{Poly}^m A$ by applying Lit ; $\text{rotaOuter} (n + n) \ n$, as defined in Sect. 3.1, composes $\text{rotaPoly} (n + n)$ with itself n times, thereby moving the n original indeterminates of e to outermost positions; the function $\text{toDChain} : \forall \{A\} \ n \rightarrow \text{Vec } A \ n \rightarrow \text{DChain } A \ n$ converts a vector to a descending chain, informally,

$$\text{toDChain} [t_2, t_1, t_0] = (\text{Lit} (\text{Lit } t_2), \text{Lit } t_1, t_0, \text{tt}) ;$$

finally, sem performs the substitution. Again, the actual code needs additional proof terms (to convince Agda that $n \leq n + n$) and type coercion (between $\text{Poly}^n (\text{Poly}^m A)$ and $\text{Poly}^{m+n} A$), which are omitted here.

3.3 Expansion

Expansion is an operation we put specific emphasis on, since it is useful when implementing cryptosystems on microprocessors with no native hardware support for arithmetic operations with polynomials or integers of cryptographic sizes. Let us use a simple yet specific example for further exposition: the polynomial expression over complex numbers $(3+2i)x^2 + (2+i)x + 1$ can be represented by $\text{Poly} (\mathbb{R} \times \mathbb{R})$, whose semantics is a function $(\mathbb{R} \times \mathbb{R}) \rightarrow (\mathbb{R} \times \mathbb{R})$. Let x be $x_1 + x_2i$, the polynomial can be expanded as below:

$$\begin{aligned} &(3 + 2i)(x_1 + x_2i)^2 + (2 + i)(x_1 + x_2i) + 1 \\ = &(3x_1^2 - 4x_1x_2 - 3x_2^2) + (2x_1^2 + 6x_1x_2 - 2x_2^2)i + (2x_1 - x_2) + (x_1 + 2x_2)i + 1 \\ = &(3x_1^2 + 2x_1 - 4x_1x_2 - x_2 - 3x_2^2 + 1) + (2x_1^2 + x_1 + 6x_1x_2 + 2x_2 - 2x_2^2)i. \end{aligned}$$

That is, a univariate polynomial over pairs, $\text{Poly} (\mathbb{R} \times \mathbb{R})$, can be expanded to $(\text{Poly}^2 \mathbb{R} \times \text{Poly}^2 \mathbb{R})$, a pair of bivariate expressions. The expansion depends on the definitions of addition and multiplication of complex numbers.

It might turn out that \mathbb{R} is represented by a fixed number of machine words: $\mathbb{R} = \text{Word}^n$. As mentioned before, in cryptosystems n could be hundreds. To compute the value of the polynomial, Poly Word^n can be further expanded to $(\text{Poly}^n \text{Word})^n$, this time using arithmetic operations defined for Word . Now that each polynomial is defined over Word , whose arithmetic operations are natively supported, we may compile the expressions, in ways discussed in Sect. 4, into a sequence of operations in assembly language. We also note that the roles played by the indeterminates x and i are of fundamental difference: x might just represent the input of the computation modelled by the polynomial expression, which will be substituted by some values at runtime, whereas i intends to model some internal (algebraic) structures and is never substituted throughout the whole computation.

Currently, such conversion and compilation are typically done by hand. We define expansion in this section and compilation in the next, as well as proving their correctness.

In general, a univariate polynomial over n -vectors, $\text{Poly} (\text{Vec } A \ n)$, can be expanded to a n -vector of n -variate polynomial, $\text{Vec} (\text{Poly}^n A) \ n$. To formally define expansion we need some helper functions. Firstly, $\text{genInd } n$ generates a vector $\text{Ind} :: \text{Lit } \text{Ind} :: \dots \text{Lit}^{n-1} \text{Ind} :: []$. It corresponds to expanding x to (x_1, x_2) in the previous example.

```

genInd : ∀ {A} n → Vec (Polyn A) n
genInd zero      = []
genInd (suc zero) = Ind :: []
genInd (suc (suc n)) = Ind :: map Lit (genInd (suc n)) .
    
```

Secondly, $\text{liftVal} : \forall \{A\} n \rightarrow A \rightarrow \text{Poly}^n A$ lifts A to $\text{Poly}^n A$ by n applications of Lit . The definition is routine.

Expansion can now be defined by:

```

expand : ∀ {A} n → Ring (Vec (Polyn A) n) → Poly (Vec A n) → Vec (Polyn A) n
expand n rv = foldP (genInd n) (map (liftVal n)) (fst rv)
    
```

For the Ind case, one indeterminate is expanded to n using genInd . For the $\text{Lit } xs$ case, $xs : \text{Vec } A \ n$ can be lifted to $\text{Vec} (\text{Poly}^n A) \ n$ by $\text{map} (\text{liftVal } n)$. For addition and multiplication, we let rv decide how to combine vectors of expressions.

The function expand alone does not say much — all the complex work is done in $rv : \text{Ring} (\text{Vec} (\text{Poly}^n A) \ n)$. To generate rv , we define the type of operations that, given arithmetic operators for A , define ring instance for vectors of A :

```

RingVec : ℕ → Set1
RingVec n = ∀ {A} → Ring A → Ring (Vec A n) .
    
```

For example, rComplex lifts arithmetic operations on A to that of complex numbers over A :

```

rComplex : RingVec 2
rComplex ((+), (×), 0, 1, neg) = ((+_c), (×_c), [0, 0], [1, 0], negC)
    
```

$$\begin{aligned}
\text{where } [x_1, y_1] +_c [x_2, y_2] &= [x_1 + x_2, y_1 + y_2] \\
[x_1, y_1] \times_c [x_2, y_2] &= [x_1 \times x_2 - y_1 \times y_2, x_1 \times y_2 + x_2 \times y_1] \\
x - y &= x + \text{neg } \mathbf{1} \times y \\
\text{negC } [x, y] &= [\text{neg } \mathbf{1} \times x_1, \text{neg } \mathbf{1} \times y] .
\end{aligned}$$

To expand a polynomial of complex numbers $\text{Poly } (\text{Vec } A \ 2)$, `expand` demands an instance of $\text{Ring } (\text{Vec } (\text{Poly}^2 A) \ 2)$. One may thus call `expand 2 (rComplex (ringP2 r))`, where $r : \text{Ring } A$. That is, we use `rComplex` to combine a pair of polynomials, designating $((:+)$, $(:\times))$ as addition and multiplication.

Correctness. Intuitively, `expand` is correct if the expanded polynomial evaluates to the same value as that of the original. To formally state the property, we have to properly supply all the needed ingredients. Consider $e : \text{Poly } (\text{Vec } A \ n)$ — a polynomial whose coefficients are vectors of length n . Let $r : \text{Ring } A$ define arithmetic operators for A , and let $\text{ringVec} : \text{RingVec } n$ define how arithmetic operators for elements are lifted to vectors. We say that `expand` is correct if, for all $xs : \text{Vec } A \ n$:

$$\begin{aligned}
\text{sem}_1 (\text{ringVec } r) e \ xs &= \text{map } (\lambda e \rightarrow \text{sem } r \ n \ e \ (\text{toDChain } xs)) \\
&\quad (\text{expand } n \ (\text{ringVec } (\text{ringP}^* r \ n)) \ e). \quad (1)
\end{aligned}$$

On the lefthand side, e is evaluated by sem_1 , using operators supplied by $\text{ringVec } r$. The value of the single indeterminant is $xs : \text{Vec } A \ n$, and the result also has type $\text{Vec } A \ n$. On the righthand side, e is expanded to $\text{Vec } (\text{Poly}^n A) \ n$, for which we need an instance of $\text{Ring } (\text{Vec } (\text{Poly}^n A) \ n)$, generated by $\text{ringVec } (\text{ringP}^* r \ n)$. Each polynomial in the vector is then evaluated individually by $\text{sem } r \ n$. The function `toDChain` converts a vector to a descending chain. The n elements in xs thus substitutes the n indeterminants of the expanded polynomial.

Interestingly, it turns out that `expand` is correct if ringVec is polymorphic — that is, the way it computes vectors out of vectors depends only on the shape of its inputs, regardless of the type and values of their elements.

Theorem 2. *For all n , $e : \text{Poly } (\text{Vec } A \ n)$, $xs : \text{Vec } A \ n$, $r : \text{Ring } A$, and $\text{ringVec} : \text{RingVec}$, property (1) holds if ringVec is polymorphic.*

Proof. Induction on e . For the base cases we need two lemmas:

- for all n , x , $es : \text{DChain } A \ n$, and r , we have $\text{sem } r \ n \ (\text{liftVal } n \ x) \ es = x$;
- for all n , $xs : \text{Vec } A \ n$, and $r : \text{Ring } A$, we have $\text{map } (\lambda e \rightarrow \text{sem } r \ n \ e \ (\text{toDChain } xs)) \ (\text{genInd } n) = xs$.

The inductive case where $e := e_1 :+ e_2$ eventually comes down to proving that (abbreviating $\lambda e \rightarrow \text{sem } r \ n \ e \ (\text{toDChain } xs)$ to sem'):

$$\begin{aligned}
\text{map } \text{sem}' \ (\text{expand } \text{ringVec } n \ e_1) \ +_{\text{VA}} \ \text{map } \text{sem}' \ (\text{expand } \text{ringVec } n \ e_2) &= \\
\text{map } \text{sem}' \ (\text{expand } \text{ringVec } n \ e_1 \ +_{\text{VP}} \ \text{expand } \text{ringVec } n \ e_2) &=
\end{aligned}$$

$\text{Ins} : \text{Set}$
 $\text{Ins} = \text{List TAC} .$

The command $\text{Const } i \ v$ stores value v in address i , $\text{Add } i \ j \ k$ fetches values stored in addresses i and j and stores their sum in address k , and similarly with Mul . Given a heap, executing an assembly program computes a new heap:

$\text{runIns} : \text{Heap} \rightarrow \text{Ins} \rightarrow \text{Heap} .$

To compile a program we employ a monad SSA , which support an operation $\text{alloc} : \text{SSA Addr}$ that returns the address of an unused cell in the heap. A naive approach is to implement SSA by a state monad that keeps a counter of the highest address that is allocated, while alloc returns the current value of the counter before incrementing it — register allocation can be performed in a separate pass. To run a SSA monad we use a function $\text{runSSA} : \forall \{A \ \text{St}\} \rightarrow \text{St} \rightarrow \text{SSA St A} \rightarrow (A \times \text{St})$ that takes a state St and yields a pair containing the result and the new state.

Compilation of a polynomial yields $\text{SSA} (\text{Addr} \times \text{Ins})$, where the second component of the pair is an assembly program, and the first component is the address where the program, once run, stores the value of the polynomial. We define compilation of $\text{Poly}^n \ \text{Word}$ by induction on n . For the base case $\text{Poly}^0 \ \text{Word} = \text{Word}$, we simply allocate a new cell and store the given value there using Const :

$\text{compile}_0 : \text{Word} \rightarrow \text{SSA} (\text{Addr} \times \text{Ins})$
 $\text{compile}_0 \ v = \text{alloc} \gg \lambda \text{addr} \rightarrow$
 $\quad \text{return} (\text{addr} , \text{Const } \text{addr} \ v :: []) .$

To compile a polynomial of type $\text{Poly}^n \ \text{Word}$, we assume that the value of the n indeterminants are already computed and stored in the heap, the locations of which are stored in a vector of n addresses.

$\text{compile} : \forall n \rightarrow \text{Vec Addr } n \rightarrow \text{Poly}^n \ \text{Word} \rightarrow \text{SSA} (\text{Addr} \times \text{Ins})$
 $\text{compile } \text{zero} \ \text{addr} = \text{compile}_0$
 $\text{compile} (\text{suc } n) (x :: \text{addr}) =$
 $\quad \text{foldP} (\text{return} (x, [])) (\text{compile } n \ \text{addr}) (\text{biOp Add}, \text{biOp Mul}) .$

In the clause for $\text{suc } n$, x is the address storing the value for the outermost indeterminant. To compile Ind , we simply return this address without generating any code. To compile $\text{Lit } e$ where $e : \text{Poly}^n \ \text{Word}$, we inductively call $\text{compile } n \ \text{addr}$. The generated code is combined by $\text{biOp } \text{op } p_1 \ p_2$, which runs p_1 and p_2 to obtain the compiled code, allocate a new address dest , before generating a new instruction $\text{op } \text{dest} \ \text{addr}_1 \ \text{addr}_2$:

$\text{biOp} : (\text{Addr} \rightarrow \text{Addr} \rightarrow \text{Addr} \rightarrow \text{TAC})$
 $\quad \rightarrow \text{SSA} (\text{Addr} \times \text{Ins}) \rightarrow \text{SSA} (\text{Addr} \times \text{Ins}) \rightarrow \text{SSA} (\text{Addr} \times \text{Ins})$
 $\text{biOp } \text{op } m_1 \ m_2 = m_1 \gg \lambda (\text{addr}_1 , \text{ins}_1) \rightarrow$
 $\quad m_2 \gg \lambda (\text{addr}_2 , \text{ins}_2) \rightarrow \text{alloc} \gg \lambda \text{dest} \rightarrow$
 $\quad \text{return} (\text{dest} , \text{ins}_1 \ ++ \ \text{ins}_2 \ ++ \ (\text{op } \text{dest} \ \text{addr}_1 \ \text{addr}_2 :: [])) .$

The following function compiles a polynomial, runs the program, and retrieves the resulting value from the heap:

```
compileRun : ∀ {n} → Vec Addr n → Addr → Polyn Word → Heap → Word
compileRun rs r0 e h =
  let ((r , ins) , _) = runSSA r0 (compile _ rs e)
  in runIns h ins !! r .
```

Correctness. Given a polynomial e , by correctness we intuitively mean that the compiled program computes the value which e would be evaluated to. A formal statement of correctness is complicated by the fact that $e : \text{Poly}^n A$ expects, as arguments, n polynomials arranged as a descending chain, each of them expects arguments as well, and ins expects their values to be stored in the heap.

Given a heap h , a chain $es : \text{DChain Word } n$, and a vector of addresses rs , the predicate $\text{Consistent } h \text{ es } rs$ holds if the values of each polynomial in es is stored in h at the corresponding address in rs . The predicate can be expressed by the following Agda datatype:

```
data Consistent (h : Heap) :
  ∀ {n} → DChain Word n → Vec Addr n → Set where
  [] : Consistent h tt []
  (::) : ∀ {n : ℕ} {es rs e r}
    → (h !! r ≡ sem n ringWord e es)
    → Consistent h es rs
    → Consistent h (e , es) (r :: rs) .
```

Observe that in the definition of $(::)$ the descending chain es is supplied to each invocation of sem to compute value of e , before e itself is accumulated to es .

The correctness of compile can be stated as:

```
compSem : ∀ (n : ℕ) {h : Heap}
  → (e : Polyn Word)
  → (es : DChain Word n)
  → (rs : Vec Addr n) → (r0 : Addr)
  → Consistent h es rs
  → NoOverlap r0 rs
  → compileRun rs r0 e h ≡ sem n e es .
```

The predicate $\text{Consistent } h \text{ es } rs$ states that the values of the descending chain es are stored in the corresponding addresses rs . The predicate $\text{NoOverlap } r_0 \text{ rs}$ states that, if an SSA monad is run with starting address r_0 , all subsequent allocated addresses will not overlap with those in rs . With the naive counter-based implementation of SSA, $\text{NoOverlap } r_0 \text{ rs}$ holds if r_0 is larger than every element in rs . The last line states that the polynomial e is compiled with argument addresses es and starting address r_0 , and the value the program computes should be the same as the semantics of e , given the descending chain es as arguments.

With all the setting up, the property $\text{compSem } n \text{ e}$ can be proved by induction on n and e .

5 Conclusions and Related Work

In dependently typed programming, a typical choice in implementing multivariate polynomials is to represent de Bruin indices using `Fin n`, the type having exactly `n` members. This is done in, for example, the `RingSolver` in the Agda standard library [5], among many. The tagless-final representation [2] is another alternative. In this paper, we have explored yet another alternative, chosen to see how far we can go in exploiting the isomorphism between $R[X_1, X_2, \dots, X_m]$ and univariate polynomial ring $R[X_1, X_2, \dots, X_{m-1}][X_m]$. It turns out that we can go quite far — we managed to represent multivariate polynomials using univariate polynomials. Various operations on them can be defined inductively. In particular, we defined how a polynomial of vectors can be expanded to a vector of polynomials, and how a polynomial can be compiled to sequences of scalar-manipulating instructions like assembly-language programs. The correctness proofs of those operations also turn out to be straightforward inductions, once we figure out how to precisely express the correctness property.

We note that the current expansion formula is provided by the programmer. For example, in order to expand a complex polynomial expression into two real ones, the programmer needs to provide (in a `RingVec`) the formula $(a_1 + b_1 i)(a_2 + b_2 i) \bmod i^2 + 1 = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + a_2 b_1) i$. We can see that the divisor polynomial of the modular relationship can actually give rise to an equational type in which $i^2 + 1 = 0$, or any pair of polynomials are considered “equal” if their difference is a multiple of the polynomial $i^2 + 1$. In the future, we would like to further automate the derivation of this formula, so the programmer will only need to give us the definition of the equational types under consideration. The `RingSolver` [5] manipulates equations into normal forms to solve them, and the solution can be used in Agda programs by reflection. It is interesting to explore whether a similar approach may work for our purpose.

Acknowledgements. The authors would like to thank the members of IFIP Working Group 2.1 for their valuable comments on the first presentation of this work.


References

1. Bird, R.S., de Moor, O.: Algebra of Programming. Prentice Hall International Series in Computer Science. Prentice Hall, Upper Saddle River (1997)
2. Carette, J., Kiselyov, O., Shan, C.-C.: Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages. *J. Funct. Program.* **19**(5), 509–543 (2009)
3. Chen, A.I.-T., Chen, C.-H.O., Chen, M.-S., Cheng, C.-M., Yang, B.-Y.: Practical-sized instances of multivariate PKCs: rainbow, TTS, and $\mathcal{H}C$ -derivatives. In: Buchmann, J., Ding, J. (eds.) *PQCrypto 2008*. LNCS, vol. 5299, pp. 95–108. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88403-3_7
4. Crockett, E., Peikert, C.: $\Lambda\lambda$: functional lattice cryptography. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24–28, pp. 993–1005. ACM (2016)

5. Danielsson, N.A.: Ring Solver, the Agda standard library. <https://github.com/agda/agda-stdlib/blob/master/src/Algebra/RingSolver.agda>
6. Hungerford, T.: Algebra. Graduate Texts in Mathematics. Springer, New York (2003). <https://doi.org/10.1007/978-1-4612-6101-8>
7. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. IACR Cryptology ePrint Archive 2012:230 (2012)
8. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-39799-X_31
9. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology (2007)



A Functional Perspective on Machine Learning via Programmable Induction and Abduction

Steven Cheung¹, Victor Darvari¹, Dan R. Ghica^{1(✉)}, Koko Muroya^{1,3},
and Reuben N. S. Rowe² 

¹ University of Birmingham, Birmingham, UK
D.R.Ghica@cs.bham.ac.uk

² University of Kent, Canterbury, UK

³ RIMS, Kyoto University, Kyoto, Japan

Abstract. We present a programming language for machine learning based on the concepts of ‘induction’ and ‘abduction’ as encountered in Peirce’s logic of science. We consider the desirable features such a language must have, and we identify the ‘abductive decoupling’ of parameters as a key general enabler of these features. Both an idealised abductive calculus and its implementation as a PPX extension of OCAML are presented, along with several simple examples.

1 A Principled Functional Language for Machine Learning

What is the right programming language for machine learning? This question can be answered in two ways. A first possible answer could take an algorithmic point of view and try to identify those constructs which are most used in machine learning programs, delivering an implementation in which the balance of various optimisation trade-offs favours such constructs. This way of answering the question has been studied quite extensively (e.g. [1,2]). A different methodological approach to this question is to first put machine learning in a logical perspective, to provide a guideline in the development of a programming language. Connecting deductive systems to computation is a preferred methodology of functional programming language design [3]. This is the methodology we follow in this paper, except we will consider inference systems beyond deduction.

The first step is, therefore, to place machine learning in a logical framework. This is a delicate, somewhat philosophical, undertaking which may be imperfect or incomplete in its representation of the extremely broad spectrum of machine-learning algorithms. However, this step must be made in order to enable the methodological machinery to crank on. We will situate our logical understanding of machine learning within C.S. Peirce’s view of deduction, induction, and abduction as the key reasoning processes in the logic of science. This view, espoused in his celebrated paper *Illustrations of the Logic of Science* is perhaps not the only

way in which the logic of machine learning, seen as a computational subsidiary to the logic of science, can be understood. In fact Peirce himself revised both his position on the role of induction and abduction, and even the terminology itself. But the clarity and elegance of his *Illustrations*, formal and conceptual, makes it a compelling organising principle.

The second step is to suggest an informal realisability-style correspondence between the logical and the computational, resulting in a programming language for machine learning in which typical algorithms can be expressed concisely and elegantly¹. This is indeed a common situation when developing functional programming paradigms. The methodological principles invoked above are incorporated into a calculus, which is then implemented as an extension of the OCAML programming language.

Contributions: We give a methodological justification for abductive inference as a logical framework for machine learning. Following an informal realisability argument for abduction we define a functional programming language for machine learning, which we implement as a PPX extension of OCAML. The language relies on a formal calculus of abduction which is studied elsewhere [4].

2 Deduction, Induction, Abduction

The division of all inference into Abduction, Deduction, and Induction may almost be said to be the Key of Logic. C.S. Peirce

We faithfully follow Peirce’s logical analysis of scientific methodology as given in his *Illustrations of the Logic of Science*. Several clarifications of terminology first. The first one is that the term ‘logic’ must not be confused with ‘deductive logic’. We are employing it in its broader sense, that of any system of formal rules employed in carrying out scientific enquiry. Similarly, the term ‘induction’ must not be confused with ‘mathematical’ or other kinds of *deductive* induction, but with the Humean principle of ‘*generalising from examples*’ [5]. Finally, the term ‘abduction’ is not used in *loc. cit.*, but the original term ‘hypothesis’ was subsequently replaced by the former, which became more popular.

Logical inference rules fall into two broad categories. Some rules, when correctly applied, result in conclusions which are at least as believable as the assumptions. They are ‘apodeictic’, i.e. beyond debate. These are the ‘deductive’ rules, the application of general principles to specific cases. Systems of deductive rules relate elegantly to functional programming via correspondences such as Kleene’s proofs-as-programs (realisability) [6] or the propositions-as-types correspondence introduced by Curry and Howard [7]. Computation carried out in such deductive functional programming languages produces definitive results. However, these ‘analytic’ rules, and the computation inspired by them, play no role in the creation of new knowledge.

¹ The Curry-Howard correspondence emphasises types, whereas realisability emphasises proofs. Because we discuss new proof rules, rather than new types, we will prefer the realisability approach.

In contrast, machine learning is a style of computation characterised by the opposite features. It is tentative, in that it produces possibly imprecise or inaccurate results. Yet it is ‘ampliative’ (or ‘synthetic’) in that it generates new knowledge. The tentative nature of the results is a necessary consequence of knowledge generation, which involves heuristics such as generalisation or guessing. The fallibility of machine learning may be unsettling, but it is an allowance we must make for the sake of creativity. The same uncertainty also characterise the synthetic logical rules of scientific discovery, induction and abduction. By formalising them we simply endow existing scientific practice with a computational dimension.

Peirce’s presentation of logical concepts is syllogistic. Deduction is the application of a Rule (‘*All men are mortal.*’) to a Case (‘*Socrates is a man.*’) in order to produce a Result (‘*Socrates is mortal.*’). In contrast, (scientific) induction is the synthesis of a Rule (‘*If the ball is struck, it moves.*’) out of a Case (‘*The ball was struck.*’) and a Result (‘*The ball moved.*’). Formalised, this rule is deductively uninteresting, $\frac{A \wedge B}{A \supset B}$. In scientific practice the rule is slightly different.

Induction either generalises from a number of samples $\frac{A \wedge B \quad \dots \quad A \wedge B}{A \supset B}$ or reinforces an existing rule in light of new evidence $\frac{A \wedge B \quad A \supset B}{A \supset B}$. The strength of the evidence can be modelled more precisely either by augmenting the logic with modalities, or quantitatively, by (frequentist) statistical inference or Bayesian belief revision. These lines of inquiry are investigated by a significant literature [8].

Abduction is the third and final arrangement of Rule, Case and Result in a distinct inference rule: given a Rule and a Result we infer the Case. The formalisation is the (deductively unsound) $\frac{B \quad A \supset B}{A}$. Peirce acknowledged abduction as the rule leading to the most uncertain, the most speculative, knowledge, but also as the rule with the potential to lead to the creation of the most interesting new knowledge. In the practice of scientific discovery, abduction is the process by which we try to answer the question ‘Why?’. This rule may seem extravagantly unsound yet it plays a crucial role in Peirce’s philosophical understanding of the logic of scientific discovery.²

More succinctly, the roles of induction and abduction can be explained as follows. Induction is a way to mechanically create models of the world from data about the world, whereas abduction is an examination of given models in order to understand why they work.

² Abduction is essential in making statements about reality when we only have access to sense-data such as measurements. For example, the Result might be ‘*The thermometer reads 10°*’ with the Rule ‘*If the temperature is 10° then the thermometer reads 10°*’. From these we can abduce the Case, that ‘*The temperature is 10°*’. Note that this can never be apodeictic because, for example, the thermometer may be broken. Denying abductive reasoning and demanding the certainty of deduction leads to universal scepticism, e.g. Descartes’s ‘*evil demon*’ which may subvert our experience of the world.

2.1 Proofs-as-Programs for Induction

The induction rule $\frac{A \wedge B \quad \cdots \quad A \wedge B}{A \supset B}$ has a natural computational interpretation as the coercion of a list of pairs of type $A \times B$ into a function $A \rightarrow B$, realisable by a collection of constants $\text{interp}_{A,B} : (A \times B) \text{ list} \rightarrow A \rightarrow B$. It is reasonable to expect the function to be both conservative, agreeing with the arguments when specified, and ampliative, supplying new and sensible values of type B when an unknown argument of type A is provided. This is interpolation.

Interpolation can only be computed for certain data types. In the most general case, if the type A is an order, i.e. it is equipped with a comparison function, then the simplest and most general interpolation method is piecewise constant interpolation. The resulting function f is constant on each interval $a_n \leq a < a_{n+1}$ with $a_n, a_{n+1} \in A$ consecutive known points, i.e. $f(a) = b_n$ for $a_n \leq a < a_{n+1}$. If A is an ordered field then the piecewise-constant interpolation can be more sensible, with the segments centred in the known values, so that for $(a_{n-1} + a_n)/2 \leq a < (a_n + a_{n+1})/2$, $f(a) = b_n$. If A is a multi-dimensional vector field then the partition of space into regions based on distance to the given points in which the function value is constant is the Voronoi tessellation [9].

If both A and B are fields then a variety of interpolation methods are available (trading off computational simplicity for precision) from linear interpolation, which approximates the function as a set of line segments, to polynomial or spline interpolations, which produce smooth functions. These methods also apply when A is a vector field ('multivariate interpolation') and even if the premises are countably many, i.e. the list of points is infinite (*streams*), via Whittaker-Shannon interpolation [10].

The alternative presentation of the induction rule, $\frac{A \wedge B \quad A \supset B}{A \supset B}$ is more subtle because its computational interpretation suggests the need to 'update' a function $A \rightarrow B$ to take into account a new pair of points $A \times B$. If we situate ourselves in the realm of approximation, we note that a function $f : A \rightarrow B$ can be always converted into a list of points $(A \times B) \text{ list}$ via *sampling*, thus reducing this rule to the previous. Concretely this would involve a family of constants $\text{samp}_{A,B} : (A \rightarrow B) \rightarrow (A \times B) \text{ list}$. A realisability-style interpretation of this rule would be more subtle because when relating lists of points and functions, interpolating then sampling at the same inputs produces the original data points, but interpolating from a set of samples in general does not produce the original function.

Finally, if we are in an approximate setting, then the computational interpretation of induction can go beyond interpolation. Interpolation is always accurate at the interpolation points, but functions can be synthesised in ways which reduce rather than avoid error at the sampling points, such as *regression* [11]. Regression is a more robust way of synthesising functions because it recognises the possibility that the sample points may incorporate noise or errors. Interpolation will *over-fit* the function to the points, a problem avoided by regression. In the next section we will see that regression plays a key role in the interpretation of abduction.

A basic ‘inductive’ core functional programming language for induction could, in principle, be designed on the basis of an applied lambda calculus with lists and special $\text{samp}_{A,B}$ and $\text{interp}_{A,B}$ constants.

2.2 Proofs-as-Programs for Abduction

Induction, computationally, may be interpreted as the synthesis of functions out of data using techniques such as interpolation or regression. This is potentially useful in the context of data science, but it is not quite machine learning. We will see how abduction fills this role. The interpretation is rather different than in the case of the induction, because in the abduction rule $\frac{B \quad A \supset B}{A}$ we will not think of A and B as data, but rather we will think of A as parameters (P) and B as a rule ($M \supset N$). This ‘higher-order’ version of abduction is particularly interesting for us: $\frac{M \supset N \quad P \supset (M \supset N)}{P}$.

The computational interpretation is as follows. Given a parametrised function $f : P \rightarrow (M \rightarrow N)$ and a non-parametric function $g : M \rightarrow N$ we want to find the values of parameter $p : P$ which makes functions $fp : M \rightarrow N$ and $g : M \rightarrow N$ be ‘as similar as possible’. One can think of g as an external phenomenon, an experiment, or an oracle, and f as a model. By abduction we need to find the best parameter values for the model so that the model instance fp best explains g . The process of ‘abducting the best parameters’ of a generic model is a general instance of a *machine learning situation*.

Concretely, a programming language would require a family of constants $\text{abd}_{P,A} : A \rightarrow (P \rightarrow A) \rightarrow P$, where A is (usually) a function type. The informal semantics of $\text{abd}mf$ is the calculation of a parameter $p : P$ so that a defined measure of distance between fp and the reference external function m is minimised. This is a generic optimisation problem which can be approached in different ways depending on the types involved.

If P , the type of the parameters, is a discrete data type then combinatorial optimisation algorithms can be used to compute p . The literature on the topic is substantial [12]. If P is a vector space then numerical approximations such as gradient descent can be used. There is an even broader literature in this area [13] going back to Cauchy’s pioneering work on numeric solutions to systems of equations. Note that if the model $P \rightarrow A$ is a smooth (differentiable) function and if the programming language has reflection [14] then gradient descent can be made very efficient by computing the differential of the function automatically [15], otherwise it can be computed numerically [16].

An abductive programming language, i.e. a simply-typed lambda calculus extended with a family of abduction primitives ($\text{abd}_{P,A}$), would offer the advantage of highly simplifying machine-learning programming by hiding the search or optimisation mechanisms from the programmer. For example, considering an oracle with signature $r : \text{float} \rightarrow \text{float}$, a linear-regression model l_c of r would be constructed as follows (in a generic functional syntax):

$$\begin{aligned}
l(p_1, p_2) x &= p_1 \times x + p_2 \\
(p'_1, p'_2) &= \text{abd}_{\text{float} \times \text{float}, \text{float} \rightarrow \text{float}} r l \\
l_c &= l(p'_1, p'_2)
\end{aligned}$$

The resulting function l_c is the concrete model, obtained from the abstract model l instantiated with abducted parameters (p'_1, p'_2) .

From the point of view of this computational interpretation we can see how induction and abduction are subtly different. Both involve the synthesis of new functions, but the mechanisms are distinct. Whereas induction synthesises a function out of data using a fixed, built-in, transparent mechanism, abduction provides the means of adjusting the parameters of a programmer-supplied function to best-fit an oracle. Induction and abduction can interact to create a reference model out of sampled data rather than using the external process directly, which would be inconvenient. The combined induction-abduction function, using a collection of data points $d : (A \times B) \text{ list}$ is defined as $\text{indabd } d f = \text{abd}_{P, A \rightarrow B} (\text{interp}_{A, B} d) f$.

There is a similarity between this style of programming and TENSORFLOW [17], a successful machine-learning library, with some differences. Our ‘abduction’ corresponds to ‘training’, but we impose no syntactic distinctions between a function used as an argument to abduction or as applied to an argument. In TENSORFLOW the programmer needs to explicitly create ‘sessions’ in which a model (‘computation graph’) can be either trained or evaluated, separately. Such distinctions are generally unpleasantly low-level.

3 Programming with Induction-Abduction

The considerations above highlight a programming idiom which from a realisability perspective relates induction and abduction with certain useful programming constructs. We are reassured by the resemblance of the inductive-abductive style of programming with established frameworks such as TENSORFLOW. We are proposing in fact an idealised version of such frameworks. The question we ask is how can a conventional functional programming language be improved or extended with inductive-abductive constructs.

Induction does not present a challenge. The `interp` family of constants are simply extrinsic functions of the requisite signature. Abduction is also introduced in the same way, but programming with it turns out to be inconvenient. The real programming language design challenge is wrestling with the bureaucratic burden of parameter management. In this section we will describe at some length our rationale for language design, with the actual language to follow. We iterate through key problems and informally present partial solutions, in order to highlight the challenge faced. The definitive solution, which addresses all these problems will be presented in the next section.

The key requirement, which will drive most of the language design, is the fact that abduction must rely on a fixed and generic optimisation algorithm. A model $P \rightarrow A \rightarrow B$ must accommodate a generic optimisation algorithm over

the space P of parameters and a norm for type B . For numeric optimisations such as gradient descent, the space P of parameters is commonly a vector space. The linear regression example becomes:

$$\begin{aligned} l &: \text{vec} \rightarrow \text{float} \rightarrow \text{float} \\ l v x &= v[0] \times x + v[1] \\ v' &= \text{indabd } r l \\ l_c &= l v' \end{aligned}$$

Implicit Parameters. Parametrising models by vectors makes for ugly syntax. Moreover, composite models must be constructed using operations which are manually lifted to manage parameters. Consider a model for confidence bounds which involves two linear functions (a simple weighted regression [18]):

$$\begin{aligned} \text{bound } v x &= (l v[0 : 1] x, l v[2 : 3] x) \\ v' &= \text{indabd } r \text{ bound} \\ \text{bound}_c &= \text{bound } v' \end{aligned}$$

where $v[m : n]$ means taking a slice of the vector v from m to n and r some suitable reference data. The model has four parameters, which must be distributed to the two linear bounds. However, there is a problem with this approach. Explicit decomposition of the parameter vector worsens the syntactic overhead. More seriously, mistakes in the slicing of the vector can lead to runtime errors which, since they involve sizes, cannot be prevented by a simple type system. Instead, we would prefer this:

$$\begin{aligned} \text{bound } x &= (l x, l x) \\ v' &= \text{indabd } r \text{ bound} \\ \text{bound}_c &= \text{bound } \{v'\} \end{aligned}$$

The vector-parameter is an implicit parameter. When a concrete model is produced from the instantiation of the abstract model with the abducted parameters, they are explicitly provided. However, managing the implicit parameter in the lifted term formers is more complex than in languages which support such feature syntactically [19].

In terms of implementation, it may seem possible to handle parameters in a monadic style, but we shall see soon why this solution would not be satisfactory when other, more subtle, requirements are taken into account.

Linear Parameters. Let us now turn to an issue that drives the ultimate design of the language, illustrated by our running examples:

$$\begin{aligned} \text{bound}_1 x &= (\lambda h. (h x, h x + 1)) l \\ \text{bound}_2 x &= (l x, l x + 1) \end{aligned}$$

What is the dimension of the parameter vector for $bound_1$ and $bound_2$? In the case of the former, it is quite obvious that the vector has two components. In the latter, it depends on whether the programmer intended the two occurrences of the function l to be abducted separately or together. We think there is a case for the parameters to be abducted together, so that both functions have two parameters, not only in an attempt to conform to a beta law which is often expected by functional programmers, but also to allow the programmer to keep control of how many parameters are independently adjustable during abduction. In this example, abduction will always lead to boundaries delimited by two lines 1 unit apart, i.e. fixed confidence bounds of a linear regression. In contrast, the weighted regression model can be defined with two separate parameter vectors (each of size two), $v_0 = [1; 0]$ and $v_1 = [1; 0]$:

$$bound\ x = (l\ \{v_0\}\ x, l\ \{v_1\}\ x)$$

In a final streamlining of the syntax, we omit parameter vectors altogether and we only indicate by $\{-\}$ that a constant is to be interpreted as a parameter, leaving the vector of parameters everywhere implicit:

$$\begin{aligned} l\ x &= \{1\} \times x + \{0\} \\ l'\ x &= \{1\} \times x + \{0\} \\ bound\ x &= (l\ x, l'\ x). \end{aligned}$$

The linearity of parameter occurrences will be always observed, so that for example terms $(\lambda x.x + x)\{0\}$ and $\{0\} + \{0\}$ are distinct, because they have one, respectively two, parameters.

In general, the parameters of a model may be contributed by both arguments and free variables, which means that parameters must be discovered not only in the body of the function and its arguments, but also in closures. Linearity and the need for ‘deep’ search of parameters indicates a simple syntactic solution to be unlikely.

In conclusion, we want our running examples to be, ideally, written as:

$$\begin{aligned} lab\ x &= a \times x + b \\ g\ x &= (l\ \{1\}\ \{0\})\ x \\ f\ x &= (l'\ \{1\}\ \{0\})\ x \\ f'\ x &= (l\ \{1\}\ \{0\})\ x \\ bound_w\ x &= (f\ x, f'\ x) \\ bound_l\ x &= (g\ x, g\ x + 1) \end{aligned}$$

where $bound_w$ is the four-parameter weighted regression model, and $bound_l$ the two-parameter unit confidence interval of a linear regression.

4 The Abductive Calculus

We have argued that the construction of a parametrised model, which is the second input of the abduction $abd_{P,A} : A \rightarrow (P \rightarrow A) \rightarrow P$, can be tedious

or erroneous in the absence of implicit parameter management. We would like parameters to be implicit in construction of a parametrised model, and to be collected as a unique and opaque type for abduction.

The previous section illustrated the construction of models with implicit parameters that are represented by specially annotated constants $\{k\}$. To enable abductive programming we propose *decoupling* as a key feature. This consists of a family of constants $\text{dec}_{V_a, A} : A \rightarrow ((V_a \rightarrow A) \times V_a)$ where V_a is an abstract type indexed by a unique name a (or ‘atom’), as a mechanism to collect implicit parameters and prepare an explicitly parametrised model.

Informally, $\text{dec } m$ computes a pair $(l : V_a \rightarrow A, p : V_a)$ by collecting all implicit parameters in $m : A$ as the vector p and turning the model m (with implicit parameters) into a parametrised model $l : V_a \rightarrow A$, a function on parameter-vectors. The name a is shared by the parameter-vector p and the parametrised model l , making type V_a unique for the model. For our leading example, where π_1 is the first projection,

$$\begin{aligned} lx &= \{1\} \times x + \{0\} \\ \text{bound } x &= (lx, lx + 1) \\ \text{bound}_p &= \pi_1(\text{dec } \text{bound}) \\ p' &= \text{indabd } r \text{ bound}_p \\ \text{bound}_c &= \text{bound}_p p' \end{aligned}$$

In this section we give an overview presentation of the *abductive calculus*, an extension of the simply-typed lambda-calculus for abductive decoupling [4]. Let \mathbb{A} be a set of names (or *atoms*). Let $(\mathbb{F}, +, -, \times, /)$ be a (fixed) field and V_a an \mathbb{A} -indexed family of opaque vector types. The types T of the calculus are defined by the grammar $T ::= \mathbb{F} \mid V_a \mid T \rightarrow T$ where $a \in \mathbb{A}$.

Terms t are defined by the grammar $t ::= x \mid \lambda x^{T'} . t \mid tt \mid k \mid t \$ t \mid \{k\} \mid A_{a, T'}(f, x).t$, where T and T' are types, f and x are variables, $\$ \in \Sigma$ binary primitive operations, $k \in \mathbb{F}$ field elements, and $a \in A$ names. The novel syntactic elements of the calculus are provisional constants $\{k\}$, which serve as implicit parameters in the above discussion, and a family of type- and name-indexed abductive decoupling functions $A_{a, T'}(f, x).t$. Decoupling functions are the implicational form of the decoupling operation dec ; they can be syntactically related by $(A_{a, T'}(f, x).t) u \equiv (\lambda(f, x).t) (\text{dec}_{V_a, T'} u)$ in the presence of tuples. We opt for the implicational form to make the scope of names explicit, as we see in the type system below.

Let $A \subset_{\text{fin}} \mathbb{A}$ be a finite set of names, Γ a sequence of typed variables $x_i : T_i$, and \mathbf{p} a sequence of elements of the field \mathbb{F} (i.e. a vector over \mathbb{F}). We write $A \vdash \Gamma$ if A is the support of Γ . The typing judgements are of shape: $A \mid \Gamma \mid \mathbf{p} \vdash t : T$, and typing derivation rules are given below.

$$\begin{array}{c} \frac{A \vdash \Gamma, T}{A \mid \Gamma, x : T \mid - \vdash x : T} \quad \frac{A \mid \Gamma, x : T' \mid \mathbf{p} \vdash t : T}{A \mid \Gamma \mid \mathbf{p} \vdash \lambda x^{T'} . t : T' \rightarrow T} \\ \frac{A \mid \Gamma \mid \mathbf{p} \vdash t : T' \rightarrow T \quad A \mid \Gamma \mid \mathbf{q} \vdash u : T'}{A \mid \Gamma \mid \mathbf{p}, \mathbf{q} \vdash tu : T} \end{array}$$

$$\begin{array}{c}
\frac{A \vdash \Gamma \quad k \in \mathbb{F}}{A \mid \Gamma \mid - \vdash \underline{k} : \mathbb{F}} \\
\\
\frac{A \mid \Gamma \mid \mathbf{p} \vdash t_1 : T_1 \quad A \mid \Gamma \mid \mathbf{q} \vdash t_2 : T_2 \quad \$: T_1 \rightarrow T_2 \rightarrow T \in \Sigma}{A \mid \Gamma \mid \mathbf{p}, \mathbf{q} \vdash t_1 \$ t_2 : T} \\
\\
\frac{A \vdash \Gamma}{A \mid \Gamma \mid \mathbf{p} \vdash \{\mathbf{p}\} : \mathbb{F}} \quad \frac{A, a \mid \Gamma, f : V_a \rightarrow T', x : V_a \mid \mathbf{p} \vdash t : T \quad A \vdash \Gamma, T', T}{A \mid \Gamma \mid \mathbf{p} \vdash \mathbf{A}_{a, T'}(f, x).t : T' \rightarrow T}
\end{array}$$

Note that the rules are linear with respect to the parameters \mathbf{p} . In a derivable judgement $A \mid \Gamma \mid \mathbf{p} \vdash t : T$, the vector \mathbf{p} gives the collection of all provisional constants from t . The decoupling function $\mathbf{A}_{a, T'}(f, x).t$ binds name a , so it requires in its typing a unique vector type V_a collecting all the provisional constants. The typing rule for the function limits the scope of the name a , so that this vector type V_a cannot be used outside of the scope of the function. As a consequence the vector type V_a is unique to the decoupling function. Variables f and x bound by the function share the type V_a but this type cannot be mixed with parameters produced by other decouplings, as they may result in vectors with different numbers of elements. This is discussed in Sect. 6.

Employing the straightforward extension by tuples (and lists) and the syntactic sugar $\text{let } x = u \text{ in } t \equiv (\lambda x.t) u$, our leading example can be written in the abductive calculus as below.

```

let l = λa.λb.λx.a × x + b in
let f = l {1} {0} in
let bound = λx.(f x, f x + 1) in
let update = A(boundp, -).boundp (indabd r boundp) in
update bound

```

Its operational semantics is specified using a variation on the Geometry of Interaction [20] which relies on graph rewriting [21]. Using this semantics the calculus is proved as sound (well-typed programs terminate with a value).

The abstract machine represents a term as a graph along the edges of which a token travels. The routing of the token is defined by language-specific rules, as are the rewrite rules. The presence of the token indicates unambiguously what rule must apply, defining in effect a particular reduction strategy.

The ‘dynamic rewriting’ style of the operational semantics has several advantages which we discuss below.

Sharing of provisional constants in the graph model is naturally represented by making provisional-constant nodes with several incoming edges. In a term model the same can only be achieved by introducing auxiliary names, a more complicated formalism. Moreover, provisional constants cannot be copied or discarded, but only shared at run-time, restrictions which are naturally reflected in a graph model, unlike in a term model.

Decoupling is a complex, dynamic runtime operation which in the graph model is surprisingly easy to formulate. Representing code and environment jointly as a single graph removes the need for complex lookups in the formulation of this rule. This is not as convenient in conventional abstract machines, because code and environment are separate, hindering the formulation of rules involving both, especially the collecting and sharing of provisional constants.

In the presence of provisional constants, a program can be interpreted in both extensional and intensional ways. For example, an extensional interpretation of $\{1\} + \{2\}$ is a value 3, while its intensional interpretation is ‘a computation of summation, given two provisional constants that are currently 2 and 3.’ The graph-rewriting abstract machine has an ability to handle both interpretations at the same time, by separating the flow of computation (the graph) and the input-output behaviour (the token).

The same ‘two-in-one’ graph representation also enables a direct proof of program equivalence by means of bisimulation, notably by dealing with the congruence property in terms of sub-graphs.

5 DecML, A Functional Language for Machine Learning

Terms in the abductive calculus can be evaluated on-line in an experimental graph-rewriting engine implementing the semantics directly.³ Additionally, we want to implement a fragment of the abductive calculus as an extension of an existing real-world functional programming language rather than as a totally new language. The major challenge of implementing the abductive calculus is to extract parameters, especially from closures. The semantic definition, which is a global reference-chasing operation similar to garbage collection, is not easily implementable. It seems to require a deep and undesirable intervention on the runtime of the language. We will therefore pursue an alternative strategy, by building a runtime structure for parameter management which is maintained by instrumentation of the native code. This will make it possible to actually maintain the model in a form in which decoupling is a trivial operation.

We implement the abductive calculus as a language extension to OCAML along with the translation from it to standard OCAML under the PPX framework [22, Sect. 7.23].⁴ In addition to OCAML terms t_{OCAML} the abductive calculus is extended with new terms $t ::= t_{\text{OCAML}} \mid [\%pc\ k] \mid [\%lift\ t]$ where, k are (floating-point) constants. If $t : A$ then $[\%model\ t] : (dict \rightarrow A) * dict$. The boundary between ‘pure’ OCAML and the abductive terms is indicated by $[\%model\ t]$, with abductive code inside the marker.

The tag $[\%model\ t]$ ensures the code t is evaluated as a term of the abductive calculus, and presents the result to the ambient OCAML code as a model with decoupled parameters. This will require a combination of syntactic transformations and runtime instrumentations. $[\%pc\ k]$ defines a provisional constant, while lifting $[\%lift\ k]$ allows identifiers from outside of the abductive fragment, including most OCAML operators, to be used as (trivial) models.

We define a translation $\lceil - \rceil$ for terms $t : A$ of the extended abductive calculus, into terms t'_{OCAML} of ‘lifted’ types $(dict \rightarrow A) * dict$ of OCAML. The first projection is the model, a function parameterised by a dictionary of parameters, which is also given, as the second projection. The translation accumulates

³ <http://bit.ly/abd-vis>.

⁴ <https://github.com/DecML/decml-ppx>.

the parameters from its sub-terms by merging their dictionaries, and it ‘lifts’ the syntactic constructs (abstraction, application, etc.) so that they match the lifted types.

The dictionary is a simple data structure that associates each provisional constant to a unique key. By using dictionaries instead of vectors, merging two sets of parameters is easy since we no longer need to consider their order inside the parametrised function. Below are the required dictionary operations in the target language: *empty* denotes an empty dictionary, *new_key* is an operation that returns a new global key, *create_dict* creates a single element dictionary from a key and a float, *lookup* returns the value that is associated with a key in a dictionary and *merge* combines two compatible dictionaries by joining them.

$$\begin{array}{ll} \textit{empty} : \textit{dict} & \textit{new_key} : \textit{unit} \rightarrow \textit{key} \\ \textit{new_dict} : \textit{key} \rightarrow \textit{float} \rightarrow \textit{dict} & \textit{lookup} : \textit{key} \rightarrow \textit{dict} \rightarrow \textit{float} \\ \textit{merge} : \textit{dict} \rightarrow \textit{dict} \rightarrow \textit{dict} & \end{array}$$

Variable x stands for any OCAML identifier, including constants, and k for a float. The translation is indexed by a set V of variables bound in the model:

$$\begin{array}{ll} [x]_V = (\textit{fst } x, \textit{snd } x) & (\text{if } x \notin V) \\ [x]_V = (\lambda_x, \textit{empty}) & (\text{if } x \in V) \\ [\%lift\ t]_V = (\lambda_t, \textit{empty}) & (\text{where } t \text{ is a pure OCAML term}) \\ [\%pc\ k]_V = (\lambda q. \textit{lookup } L\ q, \textit{new_dict } L\ k) & (\text{where } L = \textit{new_key } ()) \\ [(t_1, t_2)]_V = (\lambda q. ((F_1\ q), (F_2\ q)), \textit{merge } P_1\ P_2) & (\text{where } (F_i, P_i) = [t_i]_V) \\ [t_1\ t_2]_V = (\lambda q. ((F_1\ q)\ (F_2\ q)), \textit{merge } P_1\ P_2) & (\text{where } (F_i, P_i) = [t_i]_V) \\ [\lambda x. t]_V = (\lambda q. \lambda x. F\ q, P) & (\text{where } (F, P) = [t]_{V \cup \{x\}}) \end{array}$$

In the definition of $\%lift$, by a ‘pure’ OCAML term we mean a term with no abductive syntax or types. In concrete DECML syntax, the leading example is:

```
let (*), (+), j, z, i = [%lift (*.)], [%lift (+.)], [%pc 1.0], [%pc 0.0], [%lift 1.0] in
let l = [%model fun x → j * x + z] in
let (bp, p) = [%model fun x → (l x), (l x + i)] in
let p' = indabd r bp in
let bc = bp p' in ...
```

Induction-abduction *indabd* is not implemented as a constant, but it can be any function of the right type, implementing a generic optimisation algorithm such as gradient descent. Parameter p can be supplied to this function as an argument, if necessary, to seed the optimisation algorithm with an initial point.

This final observation also explains our decision to use the $\%model$ annotation to lift only parts of an OCAML program rather than the whole program. Inside the abductive fragment all operations are lifted to manage parameters,

which makes them less efficient and interferes with compiler optimisation. But once a model is created in the decoupled form then it can be processed in the more efficient ambient language. It is particularly important that abduction, which dominates computationally any machine learning program, can be executed natively and efficiently. The mixing of pure and instrumented code, on the other hand, can lead to subtle typing problems which could be difficult for the programmer to understand and fix. The limited access PPX has to typing information makes it a challenge to give further assistance on this matter, so a future version of this language may require substantial re-engineering.

6 Related and Further Work

Our work has been heavily influenced by `TENSORFLOW` [17]. We are aiming to provide an idealised, functional version of this framework. `DECML`, by using the decoupling mechanism, avoids the need to represent parameters using imperative state, while recognising their importance (‘variables’ in `TENSORFLOW` terminology) as a distinct language entity. We also recognise the dual-use of models, in direct and training mode, but we prefer to not make this semantic distinction syntactic. As a shallow embedding of a DSL into `PYTHON`, `TENSORFLOW` must sometimes use rather heavy-going constructs such as that of a ‘session’, which we can afford to completely elide. Moreover, by presenting a language extension rather than an embedded DSL we avoid a host of well-known problems and pitfalls [23–25].

For reasons of space and presentational focus we have also glossed over another significant distinction between inductive-abductive programming and `TENSORFLOW`. In the former, abduction is given as a fixed, language-specific, construct whereas in the latter the abduction (search and optimisation) algorithm is programmable. Of course, fixing the abduction algorithm and assuming that certain types come with fixed norms is impractical. `PROLOG` is an example of an abductive programming language in which abduction is implemented as a fixed resolution algorithm, which significantly narrows the applicability of the language to practical problems.

However, if the programming language we are extending is rich enough (such as `OCAML`), then the induction-abduction extrinsics can be simply programmed as normal library functions. The same applies to programming the norm functions explicitly. In fact a fixed abduction construct is not actually defined in the abductive calculus [4], nor is it in `DECML`! We will briefly discuss some further language design considerations for programmable abduction, which are already included in the abductive calculus but not yet implemented in `DECML`.

The key requirement is that parameters are collected as an opaque *vector* type, the key data type required by the formalisation of generic numeric optimisation problems. Since parameter collection and slicing mechanisms are complex, the dimension of abducted parameter vectors and even the order of coordinates are impossible to anticipate at compile-time. Abstracting away these details is therefore required, and any vector needs to be uniquely associated with the model

which it parametrises. Making abduction programmable also explains why we prefer to extract, rather than discard, the current values of the parameters. They are often used by programmers to seed the search and optimisation algorithms, using domain-specific knowledge.

In order to prevent erroneous uses, unique and opaque vector types must be generated for each model so that vectors produced by abduction can only be used with the original model. The opaqueness prevents access to the representation, i.e. to the bases or the individual coordinates. Only operators which are symmetric under permutations of bases will be allowed. Mathematically, they correspond to symmetric tensors, but formulated in a programmer-friendly way. This is a real, but not onerous, restriction on the way the generic optimisation algorithms are used. Indeed, if generic optimisation algorithms are to be used at all it is difficult to imagine how (or why) we may want to program them so that different axes are treated differently in the search space. As an extra bonus, linear vector operations are efficiently programmable and easily parallelisable on specialised architectures such as GPUs.

Our proposal focuses on the correspondence between inductive-abductive inference rules and programming constructs in order to extract methodological principles for the design of a machine-learning-oriented programming language. However, these correspondences are only pursued informally. A rigorous realizability definition for induction and abduction is likely to be an interesting and instructive mathematical exercise. We plan to pursue it in the future.

Besides realisability, Curry-Howard-style correspondences can also be pursued by refining the type system to distinguish between the various modalities arising out of inductive and abductive reasoning. The distinction between definite and tentative (or approximate) values can be handled by epistemic logics which distinguish between ‘known’ and ‘believed’ statements. This can lead to types for inductive-abductive programming which can track the epistemic status of results of computations. More subtly, the analytic vs. synthetic distinction can also be modelled by type systems [26] which can prove useful in the context of machine learning. This remains a longer-term project.

References

1. Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G., Olukotun, K., Ng, A.Y.: Map-reduce for machine learning on multicore. In: *Advances in Neural Information Processing Systems*, pp. 281–288 (2007)
2. King, D.E.: Dlib-ml: a machine learning toolkit. *J. Mach. Learn. Res.* **10**, 1755–1758 (2009)
3. Huet, G.: Deduction and computation. In: Bibel, W., Jorrand, P. (eds.) *Fundamentals of Artificial Intelligence: An Advanced Course*. LNCS, vol. 232, pp. 38–74. Springer, Heidelberg (1986). <https://doi.org/10.1007/BFb0022680>
4. Muroya, K., Cheung, S., Ghica, D.R.: Abductive functional programming, a semantic approach. *CoRR abs/1710.03984* (2017). Submitted for publication
5. Howson, C.: *Hume’s Problem: Induction and the Justification of Belief*. Clarendon Press, Oxford (2000)

6. Kleene, S.C.: On the interpretation of intuitionistic number theory. *J. Symb. Log.* **10**(4), 109–124 (1945)
7. Howard, W.A.: The formulae-as-types notion of construction. In: To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, vol. 44, pp. 479–490 (1980)
8. Sheridan, F.: A survey of techniques for inference under uncertainty. *Artif. Intell. Rev.* **5**(1–2), 89–119 (1991)
9. Aurenhammer, F.: Voronoi diagrams a survey of a fundamental geometric data structure. *ACM Comput. Surv. (CSUR)* **23**(3), 345–405 (1991)
10. Marks, R.: *Introduction to Shannon Sampling and Interpolation Theory*. Springer, New York (2012). <https://doi.org/10.1007/978-1-4613-9708-3>
11. Vaughn, B.K.: Data analysis using regression and multilevel/hierarchical models. *J. Educ. Meas.* **45**(1), 94–97 (2008)
12. Ehrgott, M., Gandibleux, X.: A survey and annotated bibliography of multiobjective combinatorial optimization. *OR-Spektrum* **22**(4), 425–460 (2000)
13. Snyman, J.: *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*, vol. 97. Springer, Boston (2005). <https://doi.org/10.1007/b105200>
14. Smith, B.C.: Reflection and semantics in Lisp. In: *POPL*, pp. 23–35. ACM (1984)
15. Rall, L.B.: *Automatic Differentiation: Techniques and Applications*. Springer, Heidelberg (1981). <https://doi.org/10.1007/3-540-10861-0>
16. Lyness, J.N., Moler, C.B.: Numerical differentiation of analytic functions. *SIAM J. Numer. Anal.* **4**(2), 202–210 (1967)
17. 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](https://arxiv.org/abs/1603.04467) (2016)
18. Cleveland, W.S.: Robust locally weighted regression and smoothing scatterplots. *J. Am. Stat. Assoc.* **74**(368), 829–836 (1979)
19. Lewis, J.R., Launchbury, J., Meijer, E., Shields, M.B.: Implicit parameters: dynamic scoping with static types. In: *POPL*, pp. 108–118 (2000)
20. Girard, J.Y.: Geometry of interaction I: interpretation of system F. *Stud. Log. Found. Math.* **127**, 221–260 (1989)
21. Muroya, K., Ghica, D.R.: The dynamic geometry of interaction machine: a call-by-need graph rewriter. In: *Computer Science Logic*, pp. 32:1–32:15 (2017)
22. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system release 4.02 (2013)
23. Svenningsson, J., Axelsson, E.: Combining deep and shallow embedding for EDSL. In: Loidl, H.-W., Peña, R. (eds.) *TFP 2012. LNCS*, vol. 7829, pp. 21–36. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40447-4_2
24. Scherr, M., Chiba, S.: Implicit staging of EDSL expressions: a bridge between shallow and deep embedding. In: Jones, R. (ed.) *ECOOP 2014. LNCS*, vol. 8586, pp. 385–410. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44202-9_16
25. Gibbons, J., Wu, N.: Folding domain-specific languages: deep and shallow embeddings (functional pearl). In: *ICFP*, Gothenburg, Sweden, pp. 339–347 (2014)
26. Martin-Löf, P.: Analytic and synthetic judgements in type theory. In: Parrini, P. (ed.) *Kant and Contemporary Epistemology*, pp. 87–99. Springer, Dordrecht (1994). https://doi.org/10.1007/978-94-011-0834-8_5



Polymorphic Rewrite Rules: Confluence, Type Inference, and Instance Validation

Makoto Hamana^(✉)

Department of Computer Science, Gunma University, Kiryu, Japan
hamana@cs.gunma-u.ac.jp

Abstract. We present a new framework of polymorphic rewrite rules having predicates to restrict their instances. It is suitable for formulating and analysing fundamental calculi of programming languages. A type inference algorithm and a criterion to check local confluence property of polymorphic rules are also given, with demonstration of the effectiveness of our methodology by examination of sample program calculi. It includes the call-by-need λ -calculus and Moggi's computational lambda-calculus.

1 Introduction

Fundamental calculi of programming languages are often formulated as simply-typed computation rules. Describing such a simply-typed system requires a schematic type notation that is best formulated in a *polymorphic* typed framework. To illustrate this situation, consider the simply-typed λ -calculus as a sample calculus:

$$(\beta) \quad \Gamma \vdash (\lambda x^\sigma. M) N \Rightarrow M[x := N] : \tau$$

An important point is that σ and τ are not fixed types, but schemata of types. Therefore, (β) actually describes a *family of actual computation rules*. Namely, it represents various instances of rules by varying σ and τ , such as the following.

$$\begin{array}{ll} (\beta_{\text{bool,int}}) & \Gamma \vdash (\lambda x^{\text{bool}}. M) N \Rightarrow M[x := N] : \text{int} \\ (\beta_{\text{int} \rightarrow \text{int}, \text{bool}}) & \Gamma \vdash (\lambda x^{\text{int} \rightarrow \text{int}}. M) N \Rightarrow M[x := N] : \text{bool} \end{array}$$

From the viewpoint of meta-theory, as in a mechanised formalisation of mathematics, the (β) -rule should be formulated in a polymorphic typed framework, where types τ and σ vary over simple types. This viewpoint has not been well explored in the general theory of rewriting. For instance, no method has been established for checking the *confluence property* of a general kind of polymorphically typed computation rules automatically.

We have already recognized this problem. In a previous paper [12], we investigated the decidability of various program calculi by confluence and termination checking. The type system used there was called molecular types, which was

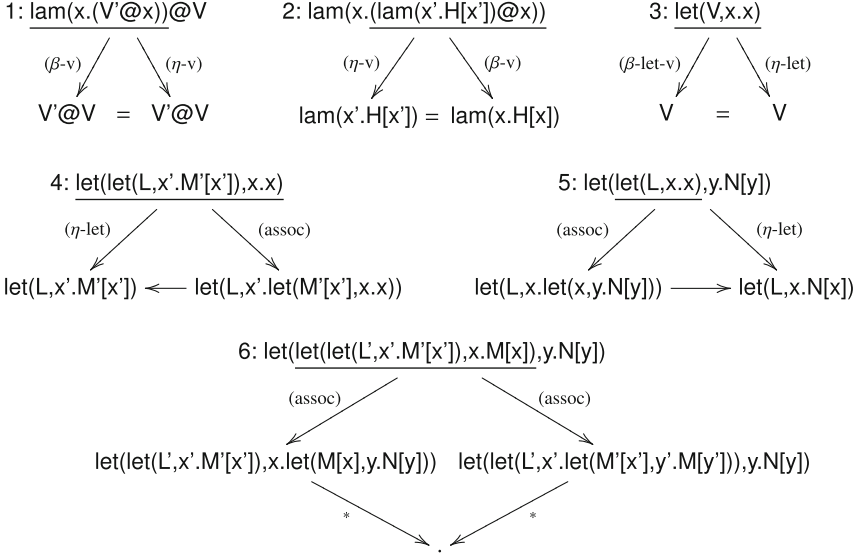


Fig. 1. Critical pairs of the λ_C -calculus

intended to mimic polymorphic types in a simple type setting. However, this mimic setting provided no satisfactory framework to address polymorphic typed rules. For example, molecular types did not have a way to vary types. Therefore, instantiation of (β) to $(\beta_{\text{bool,int}})(\beta_{\text{int} \rightarrow \text{int, bool}})$ described above could not be obtained. For that reason, no confluence of instances of polymorphic computation rules is obtained automatically. Further manual meta-theoretic analysis is necessary.

In the present paper, we give an extended framework for polymorphic computation rules that solve these issues. The framework is polymorphic and a computational refinement of second-order algebraic theories by Fiore *et al.* [5,6]. Second-order algebraic theories have been shown to be a useful framework that models various important notions of programming languages, such as logic programming [28] and algebraic effects [29], quantum computation [30]. The present polymorphic framework has also applications in these fields.

1.1 Example: Confluence of the Computational λ -Calculus

We begin with examining a sample confluence problem to illustrate our framework and methodology. We consider Moggi's computational λ -calculus, the λ_C -calculus [21], which is a fundamental λ -calculus for effectful computation. It is an extension of the call-by-value λ -calculus enriched with **let**-construct to represent sequential computation. The λ_C has two classes of terms: values and non-values.

$$\text{Values } V ::= x \mid \lambda x.M \qquad \text{Non-values } P ::= M@N \mid \text{let } x = M \text{ in } N \quad (1)$$

We now express the expression $\lambda x.M$ as $\text{lam}(x.M)$ and $\text{let } x = M \text{ in } N$ as $\text{let}(M, x.N)$. Then the computation rules of λ_C are described as follows.

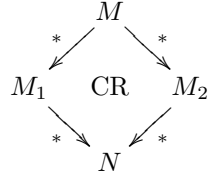
```

lamC = [rule|
  ( $\beta$ -v)    lam(x.M[x]) @ V => M[V]    ; ( $\eta$ -v)    lam(x.V @ x) => V
  ( $\beta$ -let-v) let(V, x.M[x]) => M[V]    ; ( $\eta$ -let) let(L, x.x) => L
  (let1-p)   P @ M           => let(P, x.x@M)
  (let2-v)   V @ P           => let(P, y.V@y)
  (assoc)   let(let(L, x.M[x]), y.N[y]) => let(L, x.let(M[x], y.N[y])) |]

```

The descriptions “ $\text{lamC} = [\text{rule}]$ ” and “[$]$ ” indicate the beginning and end of the rule specification in our confluence checker PolySOL. Here the metavariables V and P represent values and non-values, respectively. To the author’s best knowledge, confluence of the λ_C -calculus has not been formally proved in the literature including the original [21] and subsequent works [17, 24, 26]. We now prove confluence of the simply typed λ_C -calculus.

Confluence (CR) is a property of the reduction relation, stating that any two divergent computation paths finally commute (see the right figure). Hence, the proof requires to analyse all possible situations that admit two ways of reductions, and also to check convergence of them. In the case of λ_C , careful inspection of the rules reveals that it has, in all, 6 patterns of such situations as depicted in Fig. 1. Now we



see that all of these patterns are convergent. Importantly, this finite number of checks is sufficient to conclude that all other infinite numbers of instances of the divergent situations are convergent. This property is called *local confluence*, meaning that every possible one-step divergence is joinable. By applying Newman’s lemma [1, 13], stating “termination and local confluence imply confluence”, we can conclude that λ_C is confluent since termination of λ_C has been proved [17]. This proof method is known as Knuth and Bendix’s critical pair checking [16]. The divergent terms in Fig. 1 are called *critical pairs* because these show critical situations that may break confluence. The critical pair is obtained by an *overlap* between two rules, which is computed by *second-order unification* [20]. For example, there is an overlap between the rules (β -let-v), (η -let) because the left-hand sides of them

$$\text{let}(V, x.M[x]) \stackrel{?}{=} \text{let}(L, x.x)$$

are unifiable by second-order unification with a unifier $\theta = \{L \mapsto V, M \mapsto x.x\}$. The instantiated term $\text{let}(V, x.x)$ by θ is the source of the divergence (3:) in Fig. 1, which admits reductions by the two rules (β -let-v), (η -let).

But there are problems. In computing the critical pairs of λ_C in Fig. 1, the classical critical pair method was not applicable. Actually we need a suitable extension of the method. In the following, we list the problems, related questions and the answers we will give in this paper.

Problem 1. *The notion of unifier for an overlap is non-standard in the call-by-value case. For example, the left-hand sides of (let1-p) and (let2-v) look overlapped, but actually are not. A candidate unifier $P \mapsto V$ is not correct because P is a non-value while V is a value.*

Q1. What is a general definition of overlaps when there is a restriction on the term structures?

Problem 2. *Different occurrences of the same function symbol may have different types.*

For example, in (assoc), each **let** has actually a different type (highlighted one) as:

$$\begin{aligned} M : c \rightarrow a, N : a \rightarrow b, L : c \triangleright \\ \Gamma \vdash \text{let}^{a, (a \rightarrow b) \rightarrow b} (\text{let}^{c, (c \rightarrow a) \rightarrow a} (L, x^c.M[x]), y^a.N[y]) \\ \Rightarrow \text{let}^{c, (c \rightarrow b) \rightarrow b} (L, x^c.\text{let}^{a, (a \rightarrow b) \rightarrow b} (M[x], y^a.N[y])) \quad : b \end{aligned}$$

To compute an overlap between **let**-terms, we need also to adjust the types of **let** to equate them.

Q2. What should be the notion of unification between polymorphic second-order terms?

Moreover, specifying all the type annotations as above manually is tedious in practice. Ideally, we write a “plain” rule as (assoc), and hope that some system automatically infers the type annotations.

Q3. What is the type inference algorithm for polymorphic second-order computation rules?

In this paper, we solve these questions.

A1▶ We introduce predicates called *instance validation* on substitutions (Definition 3). It is used for formulating computation steps having some restriction of term structures. Hence they affect to the notion of critical pairs.

A2▶ We formulate the notion of unifier for polymorphic terms (Definition 7).

A3▶ We give a type inference algorithm for polymorphic computation rules in Fig. 4.

1.2 Critical Pair Checking Using the Tool PolySOL

Based on the above answers, we have implemented these features in our tool PolySOL. PolySOL is a tool to check confluence and termination of polymorphic second-order computation systems. The system PolySOL consists of about 3000 line Haskell codes, and works on the interpreter of Glasgow Haskell Compiler (tested on GHCi, version 7.6.2). PolySOL uses the feature of quasi-quotation (i.e. [signature|...] and [rule|...] are quasi-quotations) of Template Haskell [27]

with a custom parser generated by Alex (for lexer) and Happy (for parser), which realises readable notation for signature, terms and rules. It makes the language of our formal computation rules available within a Haskell script.

PolySOL first infers and checks the types of variables and terms in the computation rules using a given signature. To check confluence of the simply-typed λ_C -calculus, we declare the following signature in PolySOL:

```
siglamC = [signature|
  app : Arr(a,b),a -> b ; lam : (a -> b) -> Arr(a,b)
  let : a,(a -> b) -> b ]
```

where a, b are type variables, and $\text{Arr}(a, b)$ encodes the arrow type of the target λ -calculus in PolySOL. The rule set `lamC` given in the beginning of this subsection is actually a part of rule specification written using PolySOL's language. Using these, we can command PolySOL to perform critical pair checking.

```
*SOL> cri lamC siglamC
1: Overlap ( $\beta$ -v)-( $\eta$ -v)--- M|-> z1.(V'@z1)-----
L: lam(x.M[x])@V => M[V]
R: lam(x'.(V'@x')) => V'
      (lam(x.(V'@x))@V)
      (V'@V) <-( $\beta$ -v)- $\wedge$ -( $\eta$ -v)-> (V'@V)
      ----> (V'@V) =OK= (V'@V) <---
2: Overlap ( $\eta$ -v)-( $\beta$ -v-x)--- V|-> lam(x'.H3[x']), M'|-> z1.z2.H3[z2], V'|-> z1.z1----
L: lam(x.V@x) => V
R: (lam(x'.M'[x,x'])@V'[x]) => M'[x,V'[x]]
      (lam(x.(lam(x'.H3[x'])@x))
      lam(x'.H3[x']) <-( $\eta$ -v)- $\wedge$ -( $\beta$ -v-x)-> lam(x.H3[x])
      ----> lam(x'.H3[x']) =E= lam(x.H3[x]) <---
3: Overlap ( $\beta$ -let-v)-( $\eta$ -let)--- M'|-> V, M|-> z1.z1-----
L: let(V,x.M[x]) => M[V]
R: let(M',x'.x') => M'
      (let(V,x.x))
      V <-( $\beta$ -let-v)- $\wedge$ -( $\eta$ -let)-> V
      ----> V =OK= V <---
4: Overlap ( $\eta$ -let)-( $\beta$ -assoc)--- M|-> let(L',x'.M'[x']), N'|-> z1.z1-----
L: let(M,x.x) => M
R: let(let(L',x'.M'[x']),y'.N'[y']) => let(L',x'.let(M'[x'],y'.N'[y']))
      (let(let(L',x'.M'[x']),x.x))
      let(L',x'.M'[x']) <-( $\eta$ -let)- $\wedge$ -( $\beta$ -assoc)-> let(L',xd13.let(M'[xd13],yd13.yd13))
      ----> let(L',x'.M'[x']) =E= let(L',xd13.M'[xd13]) <---
5: Overlap ( $\beta$ -assoc)-( $\eta$ -let)--- M'|-> L, M|-> z1.z1-----
L: let(let(L,x.M[x]),y.N[y]) => let(L,x.let(M[x],y.N[y]))
R: let(M',x'.x') => M'
      (let(let(L,x.x),y.N[y]))
      let(L,x19.let(x19,y19.N[y19])) <-( $\beta$ -assoc)- $\wedge$ -( $\eta$ -let)->
      let(L,y.N[y])
      ----> let(L,x19.N[x19]) =E= let(L,y.N[y]) <---
6: Overlap ( $\beta$ -assoc)-( $\beta$ -assoc)--- L|-> let(L',x'.M'[x']), N'|-> z1.M[z1]-----
L: let(let(L,x.M[x]),y.N[y]) => let(L,x.let(M[x],y.N[y]))
R: let(let(L',x'.M'[x']),y'.N'[y']) => let(L',x'.let(M'[x'],y'.N'[y']))
      (let(let(let(L',x'.M'[x']),x.M[x]),y.N[y]))
      let(let(L',x'.M'[x']),x.let(M[x],y.N[y])) <-( $\beta$ -assoc)- $\wedge$ 
      -( $\beta$ -assoc)-> let(let(L',x.let(M'[x],y'.M[y'])),y.N[y])
      -> let(L',x.let(M'[x],y'.let(M[y'],y.N[y])))
      =E= let(L',x.let(M'[x],x'.let(M[x'],y.N[y]))) <-
#Joinable! (Total 6 CPs)
```

$$\begin{array}{c}
\frac{y : \tau \in \Gamma}{\Theta \triangleright \Gamma \vdash y : \tau} \quad \frac{(M : \sigma_1, \dots, \sigma_m \rightarrow \tau) \in \Theta \quad \Theta \triangleright \Gamma \vdash t_i : \sigma_i \quad (1 \leq i \leq m)}{\Theta \triangleright \Gamma \vdash M[t_1, \dots, t_m] : \tau} \\
\\
\frac{S \triangleright f : (\overline{\sigma_1} \rightarrow \tau_1), \dots, (\overline{\sigma_m} \rightarrow \tau_m) \rightarrow \tau \in \Sigma \quad \xi : S \rightarrow \mathcal{T} \quad \Theta \triangleright \Gamma, \overline{x_i} : \overline{\sigma_i} \vdash t_i : \sigma_i \xi \quad (\text{some } i \text{ s.t. } 1 \leq i \leq m)}{\Theta \triangleright \Gamma \vdash f^\sigma(\overline{x_1^{\overline{\sigma_1}}}.t_1, \dots, \overline{x_i^{\overline{\sigma_i}}}.t_i, \dots, \overline{x_m^{\overline{\sigma_m}}}.t_m) : \tau \xi}
\end{array}$$

Here, $\sigma \triangleq ((\overline{\sigma_1} \rightarrow \tau_1), \dots, (\overline{\sigma_m} \rightarrow \tau_m) \rightarrow \tau) \xi$.

Fig. 2. Typing rules of meta-terms

The above PolySOL’s output corresponds to the diagrams shown in Fig. 1. The labels **L**: and **R**: indicate the rules used in the left and right paths of a divergence, and the highlight in L-rule shows that the subterm is unifiable with the root of left-hand side of R-rule. For example, in the overlap 1, the subterm $\mathbf{lam}(x.M[x])$ in the L-rule is unifiable with the term $\mathbf{lam}(x'.(V'@x'))$ in the R-rule using the unifier $M \mapsto \mathbf{z1}.(V'@z1)$ described at the immediate above. Then using this information, PolySOL generates the underline term $\underline{\mathbf{lam}(x.(V'@x))}@V$ which exactly corresponds to the source in the first divergent diagram (1:) in Fig. 1. The lines involving \wedge (indicating “divergence”) mimics the divergence diagram and the joinability test in text. The sign $=OK=$ denotes syntactic equal, and $=E=$ denotes the α -equivalence.

Organisation. The paper is organised as follows. We first introduce the framework of second-order algebraic theories and computation rules in Sect. 2. We next give a type inference algorithm for polymorphic computation rules in Sect. 3. We then establish a confluence criteria based on critical pair checking in Sect. 4. In Sect. 5, we prove confluence of Maraist, Odersky, and Wadler’s call-by-need λ -calculus λ_{NEED} [18] using our framework. In Sect. 6, we summarise the paper and discuss related work.

2 Polymorphic Computation Rules

In this section, we introduce the framework of polymorphic second-order computation rules. It gives a formal unified framework to provide syntax, types, and computation for various simply-typed computational structure. It is a simplified framework of general polymorphic framework [4, 11] of second-order abstract syntax with metavariables [7] and its rewriting system [8–10] with molecular types [12]. The present framework introduces type variables into types and the feature of instance validation for instantiation of axioms. The polymorphism in this framework is essentially ML polymorphism, i.e., predicative and only universally quantified at the outermost and has type constructors on types.

Notation 1. We use the notation \overline{A} for a sequence A_1, \dots, A_n , and $|\overline{A}|$ for its length. The notation $s[u]_p$ means replacing the position p of s at with u , and $s|_p$ means selecting a subterm of the position p . We use the abbreviations “lhs” and “rhs” to mean left-hand side and right-hand side, respectively.

Types. We assume that \mathcal{A} is a set of *atomic types* (e.g. **Bool**, **Nat**, etc.), and a set \mathcal{V} of *type variables* (written as S, T, \dots). We also assume a set of *type constructors* together with arities $n \in \mathbb{N}$. A 0-ary one is regarded as a type constant. The sets of “0-order types” \mathcal{T}_0 and (at most first-order) **types** \mathcal{T} are generated by the following rules:

$$\frac{b \in \mathcal{A}}{b \in \mathcal{T}_0} \quad \frac{s \in \mathcal{V}}{s \in \mathcal{T}_0} \quad \frac{\tau_1, \dots, \tau_n \in \mathcal{T}_0 \quad T \text{ n-ary type constructor}}{T(\tau_1, \dots, \tau_n) \in \mathcal{T}_0} \quad \frac{\sigma_1, \dots, \sigma_n, \tau \in \mathcal{T}_0}{\sigma_1, \dots, \sigma_n \rightarrow \tau \in \mathcal{T}}$$

We call $\overline{\sigma} \rightarrow \tau$ with $|\overline{\sigma}| > 0$ a *function type*. We usually write types as σ, τ, \dots . A sequence of types may be empty in the above definition. The empty sequence is denoted by $()$, which may be omitted, e.g., $() \rightarrow \tau$, or simply τ . For example, **Bool** is an atomic type, $\text{List}^{(1)}$ is a type constructor, and $\text{Bool} \rightarrow \text{List}(\text{Bool})$ is a type.

Terms. A *signature* Σ is a set of function symbols of the form

$$T_1, \dots, T_n \triangleright f : (\overline{\sigma}_1 \rightarrow \tau_1), \dots, (\overline{\sigma}_m \rightarrow \tau_m) \rightarrow \tau$$

where $(\overline{\sigma}_1 \rightarrow \tau_1), \dots, (\overline{\sigma}_m \rightarrow \tau_m), \tau \in \mathcal{T}$ and type variables T_1, \dots, T_n may occur in these types. Any function symbol is of up to second-order type. A **metavariable** is a variable of (at most) first-order function type, declared as $M : \overline{\sigma} \rightarrow \tau$ (written as capital letters M, N, K, \dots). A **variable** (of a 0-order type) is written usually x, y, \dots , and sometimes written x^τ when it is of type τ . The raw syntax is given as follows.

- **Terms** have the form $t ::= x \mid x.t \mid f(t_1, \dots, t_n)$.
- **Meta-terms** extend terms to $t ::= x \mid x.t \mid f(t_1, \dots, t_n) \mid M[t_1, \dots, t_n]$.

The last form $M[t_1, \dots, t_n]$, called *meta-application*, means that when we instantiate $M : \overline{a} \rightarrow b$ with a meta-term s , free variables of s (which are of types \overline{a}) are replaced with meta-terms t_1, \dots, t_n (cf. Definition 2). We may write $x_1, \dots, x_n.t$ for $x_1 \dots x_n.t$, and we assume ordinary α -equivalence for bound variables. An equational theory is a set of proved equations deduced from a set of axioms. A metavariable context Θ is a sequence of (metavariable:type)-pairs, and a context Γ is a sequence of (variable:type in \mathcal{T}_0)-pairs. A judgment is of the form $\Theta \triangleright \Gamma \vdash t : b$. A **type substitution** $\rho : S \rightarrow \mathcal{T}$ is a mapping that assigns a type $\sigma \in \mathcal{T}$ to each type variable S in S . We write $\tau \rho$ (resp. $t \rho$) to be the one obtained from a type τ (resp. a meta-term t) by replacing each type variable in τ (resp. t) with a type using the type substitution $\rho : S \rightarrow \mathcal{T}$. A meta-term t is *well-typed* by the typing rules Fig. 2. Note that in a well-typed function term, a function symbol is annotated by its type as

$$f^\sigma(\overline{x_1^{\sigma_1}}.t_1, \dots, \overline{x_i^{\sigma_i}}.t_i, \dots, \overline{x_m^{\sigma_m}}.t_m)$$

where f has the polymorphic type $\sigma \triangleq ((\overline{\sigma}_1 \rightarrow \tau_1), \dots, (\overline{\sigma}_m \rightarrow \tau_m) \rightarrow \tau)\xi$. The type annotation is important in confluence checking of polymorphic rules. The notation $t\{x_1 \mapsto s_1, \dots, x_n \mapsto s_n\}$ denotes ordinary capture avoiding substitution that replaces the variables with terms s_1, \dots, s_n .

Definition 2 (Substitution of meta-terms for metavariables [5–7]).

Let $n_i = |\overline{\tau}_i|$ and $\overline{\tau}_i = \tau_i^1, \dots, \tau_i^{n_i}$. Suppose

$$\begin{aligned} \Theta \triangleright \Gamma', x_i^1 : \tau_i^1, \dots, x_i^{n_i} : \tau_i^{n_i} \vdash s_i : \sigma_i \quad (1 \leq i \leq k), \\ \Theta, M_1 : \overline{\tau}_1 \rightarrow \sigma_1, \dots, M_k : \overline{\tau}_k \rightarrow \sigma_k \triangleright \Gamma \vdash e : \tau \end{aligned}$$

Then the substituted meta-term $\Theta \triangleright \Gamma, \Gamma' \vdash e[\overline{M \mapsto \overline{x}.s}] : \tau$ is defined by

$$\begin{aligned} x[\overline{M \mapsto \overline{x}.s}] &\triangleq x \\ M_i[t_1, \dots, t_{n_i}][\overline{M \mapsto \overline{x}.s}] &\triangleq s_i\{x_i^1 \mapsto t_1[\overline{M \mapsto \overline{x}.s}], \dots, x_i^{n_i} \mapsto t_{n_i}[\overline{M \mapsto \overline{x}.s}]\} \\ f^\xi(\overline{y}_1.t_1, \dots, \overline{y}_m.t_m)[\overline{M \mapsto \overline{x}.s}] &\triangleq f^\xi(\overline{y}_1.t_1[\overline{M \mapsto \overline{x}.s}], \dots, \overline{y}_m.t_m[\overline{M \mapsto \overline{x}.s}]) \end{aligned}$$

where $[\overline{M \mapsto \overline{x}.s}]$ denotes a substitution for metavariables $[M_1 \mapsto \overline{x}_1.s_1, \dots, M_k \mapsto \overline{x}_k.s_k]$.

For meta-terms $\Theta \triangleright \Gamma \vdash \ell : \tau$ and $\Theta \triangleright \Gamma \vdash r : \tau$, a **polymorphic second-order computation rule** (or simply **rule**) is of the form $\Theta \triangleright \Gamma \vdash \ell \Rightarrow r : \tau$ satisfying

- (i) ℓ is a higher-order pattern [20], i.e., a meta-term in which every occurrence of meta-application in ℓ is of the form $M[x_1, \dots, x_n]$, where x_1, \dots, x_n are distinct bound variables.
- (ii) All metavariables in r appear in ℓ .

Definition 3. An **instance validation** is an arbitrary predicate valid that takes a substitution $[\overline{M \mapsto \overline{x}.s}]$ for metavariables and returns true or false. In this paper, we use the following various instance validations (but not limited to these).

- **Any:** valid $\theta \stackrel{\text{def}}{\Leftrightarrow}$ always true, i.e. any substitution is valid.
- **Injectivity:** valid $\theta \stackrel{\text{def}}{\Leftrightarrow}$ θ is an injective substitution (i.e. different metavariables must map to different meta-terms).
- **Values/non-values:** valid $[M_1 \mapsto \overline{x}_1.s_1, \dots, M_n \mapsto \overline{x}_n.s_n] \stackrel{\text{def}}{\Leftrightarrow} ((M_i \equiv V \Rightarrow s_i \text{ is a value}) \ \& \ (M_i \equiv P \Rightarrow s_i \text{ is a non-value}))$ for all $i = 1, \dots, n$.

Here, the notation “ $M_i \equiv V$ ” means that the metavariable M_i ’s letter is “V”. The definitions of values and non-values should be given separately. For the case of λ_C -calculus, we take the definition (1) for values and non-values in Sect. 1.1.

A (**polymorphic second-order computation system**) is a triple $(\Sigma, \mathcal{C}, \text{valid})$ consisting of a signature σ , a set \mathcal{C} of rules, and an instance validation valid. We write $s \Rightarrow_{\mathcal{C}} t$ to be one-step computation using $(\Sigma, \mathcal{C}, \text{valid})$ obtained

$$\begin{array}{c}
 S \text{ is the set of all type variables in } \overline{\tau}_i, \overline{\sigma}_i, \tau \quad \xi : S \rightarrow \mathcal{T} \\
 \Theta \triangleright \Gamma', \overline{x}_i : \overline{\tau}_i \vdash s_i : \sigma_i \xi \quad (1 \leq i \leq k) \quad \text{valid } [\overline{M} \mapsto \overline{x}.s] \\
 (M_1 : (\overline{\tau}_1 \rightarrow \sigma_1), \dots, M_k : (\overline{\tau}_k \rightarrow \sigma_k)) \triangleright \Gamma \vdash \ell \Rightarrow r : \tau \in \mathcal{C} \\
 \text{(RuleSub)} \frac{}{\Theta \triangleright \Gamma, \Gamma' \vdash \ell \xi [\overline{M} \mapsto \overline{x}.s] \Rightarrow_C r \xi [\overline{M} \mapsto \overline{x}.s] : \tau \xi} \\
 \\
 S \triangleright f : (\overline{\sigma}_1 \rightarrow \tau_1), \dots, (\overline{\sigma}_m \rightarrow \tau_m) \rightarrow \tau \in \Sigma \quad \xi : S \rightarrow \mathcal{T} \\
 \Theta \triangleright \Gamma, \overline{x}_i : \overline{\sigma}_i \vdash t_i \Rightarrow_C t'_i : \sigma_i \xi \quad (\text{some } i \text{ s.t. } 1 \leq i \leq m) \\
 \text{(Fun)} \frac{}{\Theta \triangleright \Gamma \vdash f^\sigma(\overline{x}_1^{\sigma_1}.t_1, \dots, \overline{x}_i^{\sigma_i}.t_i, \dots, \overline{x}_m^{\sigma_m}.t_m) \Rightarrow_C f^\sigma(\overline{x}_1^{\sigma_1}.t_1, \dots, \overline{x}_i^{\sigma_i}.t'_i, \dots, \overline{x}_m^{\sigma_m}.t_m) : \tau \xi} \\
 \\
 \text{Here, } \sigma \triangleq ((\overline{\sigma}_1 \rightarrow \tau_1), \dots, (\overline{\sigma}_m \rightarrow \tau_m) \rightarrow \tau) \xi.
 \end{array}$$

Fig. 3. Polymorphic second-order computation (one-step)

by the inference system given in Fig. 3. The (RuleSub) instantiates a polymorphic computation rule $\ell \Rightarrow r$ in \mathcal{C} by substitution $[\overline{M} \mapsto \overline{x}.s]$ of meta-terms for metavariables and substitution ξ on types, where the predicate `valid` checks whether $[\overline{M} \mapsto \overline{x}.s]$ is applicable. The (Fun) means that the computation step is closed under polymorphic function symbol contexts.

Example 4 (Decidable equality). The injectivity validation in Definition 3 is useful in formulating a system involving the notion of “names”, such as π -calculus. We define the signature Σ by

$$\triangleright \text{eq} : \text{Name}, \text{Name} \rightarrow \text{Bool} \quad \triangleright \text{a}, \text{b}, \text{c} : \text{Name} \quad \triangleright \text{false}, \text{true} : \text{Bool}$$

and the rules by

$$\begin{array}{l}
 \text{(eq1)} \quad X : \text{Name} \quad \triangleright \vdash \text{eq}(X, X) \Rightarrow \text{true} : \text{Bool} \\
 \text{(eq0)} \quad X, Y : \text{Name} \quad \triangleright \vdash \text{eq}(X, Y) \Rightarrow \text{false} : \text{Bool}
 \end{array}$$

We set the predicate `valid` to be **Injectivity**. Then we have expected computation, such as $\text{eq}(\text{a}, \text{b}) \Rightarrow_C \text{false}$, $\text{eq}(\text{b}, \text{b}) \Rightarrow_C \text{true}$. Without the instance validation, the rules become meaningless because they admit a non-injective substitution $\{X \mapsto \text{a}, Y \mapsto \text{a}\}$, which entails $\text{eq}(\text{a}, \text{a}) \Rightarrow_C \text{false}$. \blacksquare

3 Type Inference for Polymorphic Computation Rules

We have formulated that polymorphic computation rules were explicitly typed as Example 4. But when we give an implementation of confluence/termination checker, to insist that the user writes fully-annotated type and context information for computation rules is not a good system design. Hence we give a type inference algorithm. In the case of λ_C -calculus, the user only provides the signature `siglamC` and “plain” rules `lamC` in Sect. 1.1. The type inference algorithm infers the missing context and type annotations (highlights) as:

$$M : S \rightarrow T, N S \triangleright \vdash \text{app}^{\text{Arr}(S, T), S \rightarrow T} (\text{lam}^{\text{(S} \rightarrow \text{T}) \rightarrow \text{Arr}(S, T)} (x^S.M[x]), N) \Rightarrow M[N] : T$$

$\mathcal{W}(\Sigma, x)$	= if $x : \tau$ appears in Σ then $([], \triangleright x^\tau : \tau)$ else <i>error</i>
$\mathcal{W}(\Sigma, x.t)$	= let $a = \text{freshVar}$ $(\theta', \Theta \triangleright t' : \tau') = \mathcal{W}(\{x : a\} \cup \Sigma, t)$ in $(\theta', \Theta \triangleright x^a.t' : a \rightarrow \tau')$
$\mathcal{W}(\Sigma, f(\bar{t}))$	= if $f : \bar{d} \rightarrow c$ appears in Σ then let $n = \text{newNum}$ in $(\bar{d}' \rightarrow c') = \text{attach the index } n \text{ to all type vars in } (\bar{d} \rightarrow c)$ $(\theta, \Theta, \bar{u}, \bar{a}) = \text{foldr}(\mathcal{W}_{\text{iter}\Sigma})([], [], [], []) \bar{t}$ $b = \text{freshVar}$ $\theta' = \text{unify}((\bar{a} \rightarrow b)\theta, \bar{d}' \rightarrow c')$ in $(\theta' \circ \theta, \{f_n : (\bar{d}' \rightarrow c')\} \cup \Theta \triangleright f_n(\bar{u}) : b)$ else <i>error</i>
$\mathcal{W}(\Sigma, M[\bar{t}])$	= let $(\theta, \Theta, \bar{u}, \bar{a}) = \text{foldr}(\mathcal{W}_{\text{iter}\Sigma})([], [], [], []) \bar{t}$ $b = \text{freshVar}$ in $(\theta, \{M : \bar{a} \rightarrow b\} \cup \Theta \triangleright M[\bar{u}] : b)$
$\mathcal{W}_{\text{iter}\Sigma}(t, (\theta_0, \Theta_0, \bar{u}, \bar{\tau}))$	= let $(\theta, \Theta \triangleright u : \tau) = \mathcal{W}(\Sigma, t)$ in $(\theta \circ \theta_0, \Theta \cup \Theta_0, (u, \bar{u}), (\tau, \bar{\tau}))$
$\text{mkMatch}(\Theta)$	= $\{(\sigma, \tau) \mid (M : \sigma) \in \Theta, (M : \tau) \in \Theta, \sigma \neq \tau\}$
$\text{infer}(\Sigma, t)$	= let $(\theta, \Theta \triangleright u : \tau) = \mathcal{W}(\Sigma, t)$ $\theta' = \text{unify}(\text{mkMatch}(\Theta\theta)) \circ \theta$ in $\Theta\theta' \triangleright u\theta' : \tau\theta'$
$\text{infer}(\Sigma, s \Rightarrow t)$	= $\text{infer}(\{\text{rule} : s, s \rightarrow \tau\} \cup \Sigma, \text{rule}(s, t))$

- `freshVar` returns a new type variable
 - `newNum` returns a new number (or by counting up the stored number)
 - `foldr` is the usual “foldr” function for the sequence of terms (regarded as a list) to repeatedly apply the function \mathcal{W} by the function $\mathcal{W}_{\text{iter}}$
 - `unify` returns the most general unifier of the pairs of types.
 - “[]” denotes the empty sequence or substitution.
-

Fig. 4. Type inference algorithm

These annotations are important for checking confluence of polymorphic rules in computing overlapping between rules.

Algorithm. Our algorithm is given in Fig. 4, which is a modification of Damas-Milner type inference algorithm \mathcal{W} [3]. It has several modifications to cope with the language of meta-terms and to return enough type information for confluence checking. The algorithm takes a signature Σ and an un-annotated meta-term t . A sub-function \mathcal{W} returns $(\theta, \Theta \triangleright u : \tau)$, which is a pair of type substitution θ and an inferred judgment. The types in it are still need to be unified. The context

Θ may contain unifiable declarations, such as $M : \sigma$ and $M : \tau$ with $\sigma \neq \tau$, and these σ and τ should be unified. The main function $\text{infer}(\Sigma, t)$ does it, and returns the form

$$\Theta \triangleright t' : \tau.$$

The meta-term t' is a renamed t , where every function symbol f in the original t now has a unique index as f_n , and Θ is the set of inferred type declarations for f_n 's and all the metavariables occurring in t' . Similarly, for a given plain rule $s \Rightarrow t$, the function $\text{infer}(\Sigma, s \Rightarrow t)$ returns $\Theta \triangleright s' \Rightarrow t' : \tau$, where Θ is an inferred context and corresponding renamed terms s', t' as the sole term case. This is realised as inferring types for a meta-term to implement a rule using the new binary function symbol rule (see the definition of $\text{infer}(\Sigma, s \Rightarrow t)$).

We denote by $|t|$ a meta-term obtained from t by erasing all type annotations in the variables and the function symbols of t . We use the usual notion of “more general” relation on substitutions, denoted by $\tau' \geq \tau$, if there exists a substitution σ such that $\sigma \circ \tau' = \tau$.

Theorem 5 (Soundness). If $\text{infer}(\Sigma, t) = (\Theta \triangleright t' : \tau)$, then there exists Γ such that $\Theta \triangleright \Gamma \vdash t' : \tau$.

Theorem 6 (Completeness). If $\Theta \triangleright \Gamma \vdash t : \tau$ holds and $\text{infer}(\Sigma, |t|) = (\Theta' \triangleright t' : \tau')$, then $\tau' \geq \tau$ and

- If $M : \sigma \in \Theta$ then, there exists $M : \sigma' \in \Theta'$ such that $\sigma' \geq \sigma$
- If $f^{\bar{\sigma} \rightarrow \tau}$ occurs in t , then there exists $f_n : \bar{\sigma}' \rightarrow \tau' \in \Theta'$ such that f_n occurs in t' at the same position as t , and $\bar{\sigma}' \rightarrow \tau' \geq \bar{\sigma} \rightarrow \tau$.

The reason why our algorithm attaches an index n to each occurrence of a function symbol f as “ f_n ” is to distinguish different occurrences of the same f in a term, and to correctly infer the type of each of them (see Problem 2 in Sect. 1.1). If we have $\text{infer}(\Sigma, t) = (\Theta \triangleright t' : \tau)$, then we can fully annotate types for the plain term t . We can pick the type of each function symbol in t by finding $f_n : \bar{\sigma}' \rightarrow \tau' \in \Theta$, which means that this f has the inferred type $\bar{\sigma}' \rightarrow \tau'$.

4 Confluence of Polymorphic Computation Systems

In this section, we establish a confluence criterion of polymorphic computation systems based on critical pair checking. For a computation system \mathcal{C} , we regard “ $\Rightarrow_{\mathcal{C}}$ ” defined by Fig. 3 as a binary relation on well-typed terms. Moreover, we write $\Rightarrow_{\mathcal{C}}^*$ for the reflexive transitive closure, $\Rightarrow_{\mathcal{C}}^+$ for the transitive closure, and $\Leftarrow_{\mathcal{C}}$ for the converse of $\Rightarrow_{\mathcal{C}}$, respectively. We say:

1. $a, b \in A$ are *joinable*, written $a \downarrow b$, if $\exists c \in A. a \Rightarrow_{\mathcal{C}}^* c \ \& \ b \Rightarrow_{\mathcal{C}}^* c$.
2. $\Rightarrow_{\mathcal{C}}$ is *confluent* if $\forall a, b \in A. a \Rightarrow_{\mathcal{C}}^* b \ \& \ a \Rightarrow_{\mathcal{C}}^* c$ implies $b \downarrow c$.
3. $\Rightarrow_{\mathcal{C}}$ is *locally confluent* if $\forall a, b \in A. a \Rightarrow_{\mathcal{C}} b \ \& \ a \Rightarrow_{\mathcal{C}} c$ implies $b \downarrow c$.
4. $\Rightarrow_{\mathcal{C}}$ is *strongly normalising (SN)* if $\forall a \in A$, there is no infinite sequence $a \Rightarrow_{\mathcal{C}} a_1 \Rightarrow_{\mathcal{C}} a_2 \Rightarrow_{\mathcal{C}} \dots$.

We call a meta-term linear if no metavariable occurs more than once, and \mathcal{C} is *left-linear* if for every $\ell \Rightarrow r$ in \mathcal{C} , ℓ is linear.

Notion of Unifier Between Two Polymorphic Meta-Terms. To compute critical pairs, we need to compute overlapping between rules using second-order unification. Ordinary unifier between terms s and t is a substitution θ of terms for variables that makes $s\theta = t\theta$. In case of polymorphic second-order algebraic theory, we should also take into account of types. For example, what should be a unifier between the following terms?

$$\lambda^{(\text{bool} \rightarrow \top) \rightarrow \text{Arr}(\text{bool}, \top)}(x^{\text{bool}}.M[x]) \stackrel{?}{=} \lambda^{(\text{v} \rightarrow \text{int}) \rightarrow \text{Arr}(\text{v}, \text{int})}(x^{\text{v}}.\mathbf{g}^{\text{v} \rightarrow \text{int}}(x))$$

Here \top, v are type variables. These terms are unifiable by a substitution of meta-terms $\theta : M \mapsto x^{\text{bool}}.\mathbf{g}^{\text{bool} \rightarrow \text{int}}(x)$ together with a type substitution $\xi : \text{v} \mapsto \text{bool}, \top \mapsto \text{int}$. Therefore, we define:

Definition 7. A unifier between meta-terms s, t is a pair (θ, ξ) such that $s\xi\theta = t\xi\theta$, where ξ is a substitution of types for type variables, and θ is a substitution of metaterms for metavariables.

Critical Pairs of Polymorphic Computation Systems. We now formulate the notion of critical pairs for our polymorphic case. We first recall basic notions. A position p is a finite sequence of natural numbers. The empty sequence ϵ is the root position. Given a meta-term t , $t|_p$ denotes a subterm of t at a position p . Suppose a computation system \mathcal{C} is given. We say two rules $l_1 \Rightarrow r_1, l_2 \Rightarrow r_2$ in \mathcal{C} are *variant* if $l_1 \Rightarrow r_1$ is obtained by injectively renaming variables and metavariables of $l_2 \Rightarrow r_2$. We say that a position p in a term t is a *metavariable position* if $t|_p$ is a metavariable or meta-application.

Definition 8. An *overlap* between two rules $l_1 \Rightarrow r_1$ and $l_2 \Rightarrow r_2$ of a computation system $(\Sigma, \mathcal{C}, \text{valid})$ is a tuple $\langle l_1 \Rightarrow r_1, p, l_2 \Rightarrow r_2, \theta, \xi \rangle$ satisfying the following properties:

- $l_1 \Rightarrow r_1, l_2 \Rightarrow r_2$ are variants of rules in \mathcal{C} without common (meta) variables.
- p is a non-metavariable position of l_1 .
- If p is the root position, $l_1 \Rightarrow r_1$ and $l_2 \Rightarrow r_2$ are not variants.
- (θ, ξ) is a unifier between $l_1|_p$ and l_2 such that **valid** (θ) **holds**.

The component θ in an overlap is intended to be the output of the pattern unification algorithm [20]. An overlap represents an overlapping situation of computation, meaning that the term t is rewritten to the two different ways. We define $\text{overlap}(l_1 \Rightarrow r_1, l_2 \Rightarrow r_2) \triangleq \{\text{all possible overlaps between } l_1 \Rightarrow r_1 \text{ and } l_2 \Rightarrow r_2\}$. We collect all the overlaps in \mathcal{C} by $\mathcal{O} \triangleq \bigcup \{\text{overlap}(l_1 \Rightarrow r_1, l_2 \Rightarrow r_2) \mid l_1 \Rightarrow r_1, l_2 \Rightarrow r_2 \in \mathcal{C}\}$.

Definition 9. The *critical pair (CP)* generated from an overlap $\langle l_1 \Rightarrow r_1, p, l_2 \Rightarrow r_2, \theta, \xi \rangle$ is a triple $\langle r'_1, t, r'_2 \rangle$ where $t = l_1\xi\theta$ and $t \Rightarrow_{\mathcal{C}} r'_1$ which rewrites the root position of t using $l_1 \Rightarrow r_1$, and $t \Rightarrow_{\mathcal{C}} r'_2$ which rewrites the position p of t using $l_2 \Rightarrow r_2$.

Then, we obtain the critical pairs of \mathcal{C} by collecting all the critical pairs generated from overlaps in \mathcal{O} .

Proposition 10. Let $(\Sigma, \mathcal{C}, \text{valid})$ be a polymorphic second-order computation system. Suppose \mathcal{C} is left-linear. If for every critical pair $\langle t, u, t' \rangle$ of $(\Sigma, \mathcal{C}, \text{valid})$, we have $t \downarrow t'$, then $\Rightarrow_{\mathcal{C}}$ is locally confluent.

Theorem 11. Let $(\Sigma, \mathcal{C}, \text{valid})$ be a polymorphic second-order algebraic theory. Assume that \mathcal{C} is left-linear and strongly normalising. If for every critical pair $\langle t, u, t' \rangle$ of $(\Sigma, \mathcal{C}, \text{valid})$, we have $t \downarrow t'$, then $\Rightarrow_{\mathcal{C}}$ is confluent.

Proof. By Proposition 10 and Newman’s lemma.

5 Example: Confluence of the Call-by-Need λ -Calculus

We examine confluence of Maraist, Odersky, and Wadler’s call-by-need λ -calculus λ_{NEED} [18]. We consider the simply-typed version of it. The signature is:

```
sigNeed = [signature|
  lam : (a->b) -> Arr(a,b); app : Arr(a,b), a -> b; let : a, (a->b) -> b ]|
```

which consists of the function symbol `lam` for λ -abstraction, `app` (also written as an infix operator `@` below) for application, and `let`-construct. We represent the arrow types of the “object-level” λ_{NEED} -calculus by the binary type constructor `Arr`, and use the function types `a->b` of the “meta-level” polymorphic second-order computation system to represent binders, where `a, b` are type variables. The λ_{NEED} has five rules, which are straightforwardly defined in PolySOL as:

```
lmdNeed = [rule|
  (rG)   let(M, x.N)                => N
  (rI)   lam(x.M[x]) @ N            => let(N, x.M[x])
  (rV-v) let(V, x.C[x])            => C[V]
  (rC-v) let(V, x.M[x])@N          => let(V, x.M[x]@N)
  (rA)   let(let(L, x.M[x]), y.N[y]) => let(L, x.let(M[x], y.N[y])) ]|
```

We also impose the distinction of values and non-values as in the $\lambda_{\mathcal{C}}$ -calculus. Here `V` is a metavariable for values, and `M, N, C, L` are metavariables for all terms. We choose the instance validation `validto` to be **Values/non-values** in Definition 3. We can tell it to PolySOL by writing the suffix “-v” in the labels `(rV-v)`, `(rC-v)`. We command PolySOL to perform critical pair checking.

```
*PolySOL> cri lmdNeed sigNeed
1: Overlap (rG)-(rA)--- M|-> let(L',x'.M'[x']), N'|-> z1.N'-----
L: let(M,x.N) => N
R: let(let(L',x'.M'[x']),y'.N'[y']) => let(L',x'.let(M'[x'],y'.N'[y']))
      let(let(L',x'.M'[x']),x.N)
      N <-(rG)-^(rA)-> let(L',xd2.let(M'[xd2],yd2.N))
      ----> N =OK= N <---
2: Overlap (rC-v)-(rG)--- M'|-> V, M'|-> z1.N'-----
L: let(V,x.M[x])@N => let(V,x.(M[x]@N))
R: let(M',x'.N') => N'
```

```

                                (let(V,x.N')@N)
let(V,x5.(N'@N)) <-(rC-v)-^-(rG)-> (N'@N)
                                ----> (N'@N) =OK= (N'@N) <----
3: Overlap (rC-v)-(rV-v)--- V'|-> V, C'|-> z1.M[z1]-----
L: let(V,x.M[x])@N => let(V,x.(M[x]@N))
R: let(V',x'.C'[x']) => C'[V']

                                (let(V,x.M[x])@N)
let(V,x7.(M[x7]@N)) <-(rC-v)-^-(rV-v)-> (M[V]@N)
                                ----> (M[V]@N) =OK= (M[V]@N) <----
4: Overlap (rA)-(rG)--- M'|-> L, M|-> z1.N'-----
L: let(let(L,x.M[x]),y.N[y]) => let(L,x.let(M[x],y.N[y]))
R: let(M',x'.N') => N'

                                let(let(L,x.N'),y.N[y])
let(L,x10.let(N',y10.N[y10])) <-(rA)-^-(rG)-> let(N',y.N[y])
                                ----> let(N',y10.N[y10]) =E= let(N',y.N[y]) <----
5: Overlap (rA)-(rA)--- L|-> let(L',x'.M'[x']), N'|-> z1.M[z1]-----
L: let(let(L,x.M[x]),y.N[y]) => let(L,x.let(M[x],y.N[y]))
R: let(let(L',x'.M'[x']),y'.N'[y']) => let(L',x'.let(M'[x'],y'.N'[y']))

                                let(let(let(L',x'.M'[x']),x.M[x]),y.N[y])
let(let(L',x'.M'[x']),x.let(M[x],y.N[y])) <-(rA)-^
                                -(rA)-> let(let(L',x'.let(M'[x'],y'.M[y'])),y.N[y])
----> let(L',x.let(M'[x],y.let(M[y],y.N[y])))
                                =E= let(L',x'.let(M'[x'],x.let(M[x],y.N[y]))) <----
#Joinable! (Total 5 CPs)

```

PolySOL reports that there are 5 critical pairs, and all are successfully joinable. This shows local confluence. Strong normalisation of λ_{NEED} can be shown by a translation as the treatment of λ_{C} in [24]. Hence we conclude that λ_{NEED} is confluent. In [18], confluence of untyped λ_{NEED} has been established by a different proof method, i.e., analysis of developments steps in the λ -calculus [2]. This method is somewhat specific to the case of λ -calculus. In contrast to it, our approach is general rewriting theoretic, and not specific to variants of λ -calculus, i.e., based on critical pair checking of computation rules.

6 Summary and Related Work

6.1 Summary

We have presented a new framework of polymorphic computation rules having predicates to restrict their instances. It was shown to be suitable for formulating and analysing fundamental calculi of programming languages. We have given a type inference algorithm and a criteria to check confluence property of polymorphic rules. These have given a handy method to prove confluence of second-order computation rules. We have demonstrated effectiveness of our methodology by examining sample program calculi using our framework.

6.2 Related Work

Nipkow has studied critical pairs for confluence of higher-order rewrite systems [19, 23]. The rewrite rule format there was rules on simply-typed λ -terms modulo $\beta\eta$ -equivalence, and there are *no* polymorphic types nor instance validation. Therefore, none of our examples (λ_{C} , λ_{NEED} , and decidable equality) and the

confluence shown in the present paper can be directly formulated and checked in Nipkow's framework.

There are systems of automatic checking of confluence of Nipkow's higher-order rule format, such as ACPH [25] and CSI^{ho} [22]. Because these are based on Nipkow's format, these tools do not have the features of polymorphic types nor instance validation. Hence, all of our examples are beyond the scope of the existing confluence checking systems. In this respect, to the best of the author's knowledge, our system is the first automatic tool that can check confluence of the call-by-value variants of the λ -calculus directly, as shown in the paper.

A framework of polymorphic higher-order rewrite rules was given [14, 15], and our framework is similar but there are several differences in the foundations, e.g., our framework is based on (polymorphic) second-order algebraic theories [4, 6], while theirs is based on a polymorphic λ -calculus. The main purpose of [14, 15] was to establish termination criterion of higher-order rewrite rules. The issue of confluence of polymorphic rules has remained untouched. To the best of the author's knowledge, the present paper is probably the first study of confluence of general kind of polymorphic second-order rewrite rules (N.B. this is not about the polymorphic λ -calculus).

We gave a type inference algorithm of polymorphic computation rules, which has not been given in the context of rewriting theory (while it may be standard in the context of the theory of programming languages). Lack of a suitable type inference algorithm in the theory of rewriting has affected the existing higher-order confluence tools such as CSI^{ho} and ACPH. These tools force the user to write more detailed type information in rewrite rule specifications than PolySOL. The user needs to declare all free and bound variables with their types used in the rules. PolySOL's rule specification is simpler due to the type inference. Our algorithm may also be beneficial to other tools to improve this situation.

In [12], the present author developed a simply-typed framework of second-order equational logic and computation rules, and gave a tool SOL for checking methods of termination confluence. It lacked proper polymorphism and instance validation, hence the examples considered in the present paper could not be handled. Note that Plotkin's call-by-value λ -calculus could be formulated in [12] by explicitly modifying the rules by meta-programming like method to reflect value variables. But this approach works only for small calculi, such as the call-by-value λ -calculus, and for larger calculi, such as λ_C and λ_{NEED} , the number of overlaps explodes and we cannot hope to manage it.

In [4], we gave a general framework of multiversal polymorphic algebraic theories and their algebraic models. It admits multiple type universes and higher-kinded polymorphic types, hence it is richer than the present setting. The model theory encompasses the present simpler framework. But in [4], we did not develop neither polymorphic computation rules, confluence, nor instance validation.

Acknowledgments. I am grateful to Masahito Hasegawa for his question on how to check confluence of Moggi's computational λ -calculus by my previous tool SOL. It opened my eyes to the necessity of proper treatment of call-by-value calculi and led me to the notion of instance validation developed in this paper.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
2. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. North Holland, Amsterdam (1984)
3. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of POPL 1982, pp. 207–212 (1982)
4. Fiore, M., Hamana, M.: Multiversal polymorphic algebraic theories: syntax, semantics, translations, and equational logic. In: 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, pp. 520–529 (2013)
5. Fiore, M., Hur, C.-K.: Second-order equational logic (Extended Abstract). In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 320–335. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15205-4_26
6. Fiore, M., Mahmoud, O.: Second-order algebraic theories. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 368–380. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15155-2_33
7. Hamana, M.: Free Σ -monoids: a higher-order syntax with metavariables. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 348–363. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30477-7_23
8. Hamana, M.: Universal algebra for termination of higher-order rewriting. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 135–149. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_11
9. Hamana, M.: Higher-order semantic labelling for inductive datatype systems. In: Proceedings of PPDP 2007, pp. 97–108. ACM Press (2007)
10. Hamana, M.: Semantic labelling for proving termination of combinatory reduction systems. In: Escobar, S. (ed.) WFLP 2009. LNCS, vol. 5979, pp. 62–78. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11999-6_5
11. Hamana, M.: Polymorphic abstract syntax via grothendieck construction. In: Hofmann, M. (ed.) FoSSaCS 2011. LNCS, vol. 6604, pp. 381–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19805-2_26
12. Hamana, M.: How to prove your calculus is decidable: practical applications of second-order algebraic theories and computation. Proc. ACM Program. Lang. **1**(22), 1–28 (2017)
13. Huet, G.: Confluent reductions: abstract properties and applications to term rewriting systems. J. ACM **27**(4), 797–821 (1980)
14. Jouannaud, J.-P., Rubio, A.: Polymorphic higher-order recursive path orderings. J. ACM **54**(1), 2:1–2:48 (2007)
15. Jouannaud, J.-P., Rubio, A.: Normal higher-order termination. ACM Trans. Comput. Log. **16**(2), 13:1–13:38 (2015)
16. Knuth, D., Bendix, P.: Simple word problems in universal algebras. In: Computational Problem in Abstract Algebra, pp. 263–297. Pergamon Press, Oxford (1970)
17. Lindley, S., Stark, I.: Reducibility and $\top\top$ for computation types. In: Proceedings of TLCA 2005, pp. 262–277 (2005)
18. Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. J. Funct. Program. **8**(3), 275–317 (1998)
19. Mayr, R., Nipkow, T.: Higher-order rewrite systems and their confluence. Theor. Comput. Sci. **192**(1), 3–29 (1998)
20. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. J. Log. Comput. **1**(4), 497–536 (1991)

21. Moggi, E.: Computational lambda-calculus and monads. LFCS ECS-LFCS-88-66, University of Edinburgh (1988)
22. Nagele, J., Felgenhauer, B., Middeldorp, A.: CSI: new evidence – a progress report. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 385–397. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_24
23. Nipkow, T.: Higher-order critical pairs. In: Proceedings of 6th IEEE Symposium Logic in Computer Science, pp. 342–349 (1991)
24. Ohta, Y., Hasegawa, M.: A terminating and confluent linear lambda calculus. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 166–180. Springer, Heidelberg (2006). https://doi.org/10.1007/11805618_13
25. Onozawa, K., Kikuchi, K., Aoto, T., Toyama, Y.: ACPH: System description. In: 6th Confluence Competition (CoCo 2017)(2017)
26. Sabry, A., Wadler, P.: A reflection on call-by-value. ACM Trans. Program. Lang. Syst. **19**(6), 916–941 (1997)
27. Sheard, T., Jones, S.P.: Template metaprogramming for Haskell. In: Proceedings of Haskell Workshop 2002 (2002)
28. Staton, S.: An algebraic presentation of predicate logic. In: Pfenning, F. (ed.) FoSSaCS 2013. LNCS, vol. 7794, pp. 401–417. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37075-5_26
29. Staton, S.: Instances of computational effects: an algebraic perspective. In: Proceedings of LICS 2013, p. 519 (2013)
30. Staton, S.: Algebraic effects, linearity, and quantum programming languages. In: Proceedings of POPL 2015, pp. 395–406 (2015)



Confluence Modulo Equivalence with Invariants in Constraint Handling Rules

Daniel Gall^(✉) and Thom Frühwirth

Institute of Software Engineering and Programming Languages,
Ulm University, 89069 Ulm, Germany
{daniel.gall,thom.fruehwirth}@uni-ulm.de

Abstract. Confluence denotes the property of a state transition system that states can be rewritten in more than one way yielding the same result. Although it is a desirable property, confluence is often too strict in practical applications because it also considers states that can never be reached in practice. Additionally, sometimes states that have the same semantics in the practical context are considered as different states due to different syntactic representations. By introducing suitable invariants and equivalence relations on the states, programs may have the property to be confluent modulo the equivalence relation w.r.t. the invariant which often is desirable in practice.

In this paper, a sufficient and necessary criterion for confluence modulo equivalence w.r.t. an invariant for Constraint Handling Rules (CHR) is presented. It is the first approach that covers invariant-based confluence modulo equivalence for the de facto standard semantics of CHR. There is a trade-off between practical applicability and the simplicity of proving a confluence property. Therefore, a better manageable subset of equivalence relations has been identified that allows for the proposed confluence criterion and simplifies the confluence proofs by using well established CHR analysis methods.

1 Introduction

In program analysis, the *confluence* property of a program plays an important role. It ensures that any computation for a given start state results in the same final state. Hence, if more than one rule is applicable in a state it does not matter which rule is chosen.

Constraint Handling Rules (CHR) is a declarative programming language that has its origins in constraint logic programming [1]. Confluence analysis has been studied for CHR for a long time [2–4].

While it is a desirable property, in practical applications confluence is often too strict. For instance, it requires even states that can never be reached in a practical context to satisfy the confluence property. Therefore, *invariant-based confluence* [5–7] has been established. It only considers states that satisfy a user-defined invariant, whereas standard confluence analysis considers even states

that are invalid and cannot appear at runtime. With invariant-based confluence analysis it is possible to exclude those states from the confluence analysis as long as the rules of the program maintain the invariant.

Another method of making the confluence property available for more practical programs is to define an equivalence relation on states. A program is *confluent modulo a (user-defined) equivalence relation* if all states in the same equivalence class lead to final states of the same equivalence class [8,9]. In many programs, some states can be considered as equivalent with respect to a user-defined equivalence relation, although their actual representation in the program differs. For example, if sets of numbers are represented as lists, all states with permutations of the same list represent the same set and it might be reasonable to consider them equivalent. Hereby, confluence modulo equivalence can be used to show that for the same start state a program yields the same set as a result, although the actual representation as a list might differ.

There is a trade-off between the applicability in practical contexts and the simplicity of proving a confluence property: There is a decidable, sufficient and necessary criterion for strict confluence of terminating CHR programs [1]. When adding invariants, decidability of the criterion is lost depending on the invariant. For confluence modulo equivalence, the proofs become even harder as all states in the same equivalence class have to be considered.

In this paper, a sufficient and necessary criterion for invariant-based confluence modulo a user-defined equivalence is presented. For this purpose, a class of well-behaving equivalence relations is identified for which the proposed criterion can be applied. The confluence criterion is directly available for the equivalence-based operational semantics of CHR [7,10] that is the de facto standard of CHR semantics. By a running example it is shown that the defined class of equivalence relations is meaningful in a sense that it contains a non-trivial equivalence relation that satisfies its restrictions. Further examples have been tried indicating that the approach is promising to be more widely applicable.

In our approach, we use CHR in the pure form. We then restrict the equivalence relations to a meaningful class and present a formal proof method for invariant-based confluence modulo equivalence.

The contributions of the paper are

- the identification of a class of equivalence relations (called *compatible* equivalence relations) that maintains the monotonicity property of CHR and therefore allows for a confluence analysis based on rule states and overlaps of rules (c.f. Sect. 3),
- a sufficient and necessary criterion for an invariant-based confluence modulo equivalence for terminating CHR programs with a decidable invariant and a compatible equivalence relation (c.f. Sect. 4), and
- the application of this approach in a non-trivial running example.

Our approach is the first that covers invariant-based confluence modulo equivalence for the standard semantics of CHR. Other approaches either only consider invariants without user-defined equivalence relations [5–7] or use a special-purpose operational semantics of CHR that is claimed to extend the

standard semantics [8,9]. The latter approach introduces a meta-level to prove confluence modulo equivalence. In contrast to the meta-level proof method, the confluence criterion in this paper uses well-established standard notions of CHR states and analysis methods.

The paper is structured as follows: In Sect. 2 the preliminaries necessary for understanding the paper are given. For this purpose, definitions of confluence modulo equivalence, Constraint Handling Rules and some program analysis methods for CHR are recapitulated. Then, the class of equivalence relations regarded in this paper is defined in Sect. 3. The proof method for invariant-based confluence modulo equivalence is given in Sect. 4. The results and their relation to existing work are discussed in Sect. 5.

2 Preliminaries

We recapitulate the basic notions of confluence modulo equivalence, give a brief introduction to CHR and some program analysis techniques and summarize the established results for (invariant-based) confluence in CHR.

2.1 Confluence Modulo Equivalence

The notion of confluence modulo equivalence is defined for general state transition systems in this section.

Definition 1 (state transition system). *A state transition system is a tuple (Σ, \mapsto) where Σ is an arbitrary (possibly infinitely large) set of states and $\mapsto \subseteq \Sigma \times \Sigma$ is a transition relation over the states. By \mapsto^* we denote the reflexive transitive closure of \mapsto .*

Informally, confluence modulo equivalence means that all possible computations in a transition system starting in equivalent states finally lead to equivalent states again. We then call two states from those different computations joinable. This is illustrated in Fig. 1.

$$\begin{array}{ccc} \sigma_1 \mapsto^* \sigma_2 \mapsto^* \tau & & \\ \approx & & \approx \\ \sigma'_1 \mapsto^* \sigma'_2 \mapsto^* \tau' & & \end{array}$$

Fig. 1. Confluence modulo equivalence

Definition 2 (joinability modulo equivalence). *In a state transition system (Σ, \mapsto) two states $\sigma, \sigma' \in \Sigma$ are joinable modulo an equivalence relation \approx if and only if $\exists \tau, \tau' \in \Sigma . \sigma \mapsto^* \tau \wedge \sigma' \mapsto^* \tau' \wedge \tau \approx \tau'$. We then write $\sigma \downarrow \approx \sigma'$. If \approx is the identity equivalence relation $=$, we write $\sigma \downarrow \sigma'$ and say that σ and σ' are joinable.*

Definition 3 (confluence modulo equivalence [11]). A state transition system (Σ, \mapsto) is confluent modulo an equivalence relation \approx , if and only if for all $\sigma_1, \sigma'_1, \sigma_2, \sigma'_2 : (\sigma_1 \approx \sigma'_1) \wedge (\sigma_1 \mapsto^* \sigma_2) \wedge (\sigma'_1 \mapsto^* \sigma'_2) \rightarrow (\sigma_2 \downarrow \approx \sigma'_2)$.

If \approx is the state equivalence relation $=$, confluence modulo $=$ coincides with basic confluence. For terminating transition systems, it suffices to show *local confluence*, as we will see in the following definition and theorem.

Definition 4 (local confluence [11]). A state transition system (Σ, \mapsto) has the α and β property w.r.t. an equivalence relation \approx if and only if it satisfies the α and β conditions, respectively:

$$\begin{aligned} \alpha: & \forall \sigma, \tau, \tau' \in \Sigma : \sigma \mapsto \tau \wedge \sigma \mapsto \tau' \rightarrow \tau \downarrow \approx \tau'. \\ \beta: & \forall \sigma, \tau, \tau' \in \Sigma : \sigma \mapsto \tau \wedge \sigma \approx \tau' \rightarrow \tau \downarrow \approx \tau'. \end{aligned}$$

A state transition system is locally confluent modulo an equivalence relation \approx if and only if it has the α and the β property.

Note that in the rewriting literature, the α property is also known as *local confluence modulo equivalence* and the β property as *local coherence modulo equivalence* [12]. In this paper, *local confluence modulo equivalence* requires both the α and the β property.

In the theorem of Huet [11] it is shown that local confluence modulo an equivalence relation \approx implies confluence modulo \approx for terminating transition systems.

Theorem 1 (Huet [11]). Let (Σ, \mapsto) be a terminating transition system. For any equivalence \approx , (Σ, \mapsto) is confluent modulo \approx if and only if (Σ, \mapsto) is locally confluent modulo \approx .

2.2 Constraint Handling Rules

We now define the state transition system of CHR. We begin with CHR states.

Definition 5 (CHR state). A CHR state is a tuple $\langle \mathbb{G}; \mathbb{C}; \mathbb{V} \rangle$ where the goal \mathbb{G} is a multi-set of constraints, the built-in constraint store \mathbb{C} is a conjunction of built-in constraints and \mathbb{V} is a set of global variables. All variables occurring in a state that are not global are called local variables [7, p. 33 et seq., Definition 8.1]. If the contents of \mathbb{C} and \mathbb{V} are empty, irrelevant or clear from the context, we use a short-hand notation where only the constraints in \mathbb{G} are enumerated.

CHR states can be modified by rules that together form a CHR program.

Definition 6 (CHR program). A CHR program is a finite set of so-called simpagation rules of the form $r : H_k \setminus H_r \Leftrightarrow G \mid B_c, B_b$ where r is an optional rule name, the heads H_k and H_r are multi-sets of CHR constraints, the guard G is a conjunction of built-in constraints and the body is a multi-set of CHR constraints B_c and a conjunction of built-in constraints B_b . If G is empty, it is interpreted as the built-in constraint \top .

We introduce short forms for the following special cases:

Simplification Rules. If $H_k = \emptyset$, we also write $H_r \Leftrightarrow G \mid B_c, B_b$.

Propagation Rules. If $H_r = \emptyset$, we also write $H_k \Rightarrow G \mid B_c, B_b$.

Informally, a rule is applicable, if the heads match constraints from the goal store \mathbb{G} and the guard holds, i.e. is a consequence of the built-in constraints \mathbb{C} . In that case, the state is rewritten: The constraints matching the part H_r of the head are removed and the constraints matching H_k are kept. The user-defined body constraints B_c are added to the goal store \mathbb{G} , the built-in body constraints B_b and the constraints from the guard G are added to the built-in store \mathbb{C} .

Example 1 (Multi-Set Items [9]). Consider the following small CHR program, that collects items represented in individual *item/1* constraints to a multi-set represented by a constraint of the form *mset(L)* where L is a list of items. The program has the following rule:

$$mset(L), item(A) \Leftrightarrow mset([A|L]).$$

For the initial constraint store $item(a), item(b), mset([])$ the program can apply the rule on $mset([])$ and $item(b)$ which results in the constraint store $mset([b]), item(a)$. The rule can be applied again to this state, resulting in the constraint store $mset([a, b])$. However, the same program can also yield the constraint store $mset([b, a])$. Hence, the program is not confluent. In the following, we will return to this running example and provide an invariant and equivalence relation together with a proof method to show that the program is actually confluent modulo the equivalence relation w.r.t. the invariant.

In the context of the operational semantics, we assume a constraint theory \mathcal{CT} for the interpretation of the built-in constraints. We define an equivalence relation over CHR states.

Definition 7 (state equivalence [7, 10]). Let $\rho_i := \langle \mathbb{G}_i; \mathbb{C}_i; \mathbb{V}_i \rangle$ for $i = 1, 2$ be two CHR states with local variables \bar{y}_1, \bar{y}_2 that have been renamed apart. $\rho_1 \equiv \rho_2$ if and only if $\mathcal{CT} \models \forall (\mathbb{C}_1 \rightarrow \exists \bar{y}_2. ((\mathbb{G}_1 = \mathbb{G}_2) \wedge \mathbb{C}_2)) \wedge \forall (\mathbb{C}_2 \rightarrow \exists \bar{y}_1. ((\mathbb{G}_1 = \mathbb{G}_2) \wedge \mathbb{C}_1))$ where $\forall F$ is the universal closure of formula F and $=$ is syntactical equivalence. The equivalence class of a CHR state is defined as $[\rho] := \{\rho' \mid \rho' \equiv \rho\}$.

Example 2 (state equivalence). By the above definition of state equivalence, the following states are equivalent [7, p. 34]:

- $\langle c(X); \top; \emptyset \rangle \equiv \langle c(Y); \top; \emptyset \rangle$, i.e. local variables can be renamed.
- $\langle c(X); X = 0; \{X\} \rangle \equiv \langle c(0); X = 0; \{X\} \rangle$, i.e. variable bindings from the built-in store can be applied to the goal store.
- $\langle \emptyset; X = Y \wedge Y = 0; \emptyset \rangle \equiv \langle \emptyset; X = 0 \wedge Y = 0; \emptyset \rangle$, i.e. equivalent built-in stores can be interchanged.
- $\langle c(0); \top; \{X\} \rangle \equiv \langle c(0); \top; \emptyset \rangle$, i.e. unused global variables can be omitted.
- However, $\langle c(X); \top; \{X\} \rangle \not\equiv \langle c(Y); \top; \{Y\} \rangle$, i.e. X and Y are free variables and therefore the logical readings of the states are different. Global variables can be used to bridge information between two states.

The operational semantics is now defined by the following transition scheme over equivalence classes of CHR states.

Definition 8 (operational semantics of CHR [7,10]). *For a rule r , the variables appearing in the rule are called local variables. A variant of a rule is a copy of a rule where a subset of its local variables has been renamed.*

For a CHR program, the state transition system over CHR states and the rule transition relation \mapsto is defined as the following transition scheme:

$$\frac{r : H_k \setminus H_r \Leftrightarrow G \mid B_c, B_b}{[(H_k \uplus H_r \uplus \mathbb{G}; G \wedge \mathbb{C}; \mathbb{V})] \mapsto^r [(H_k \uplus B_c \uplus \mathbb{G}; G \wedge B_b \wedge \mathbb{C}; \mathbb{V})]}$$

Thereby, r is a variant of a rule in the program such that its local variables are disjoint from the variables occurring in the representative of the pre-transition state. We may just write \mapsto instead of \mapsto^r if the rule r is clear from the context.

From now on, we only consider equivalence classes of CHR states, since the state transition system is defined over equivalence classes.

Example 3. In this example, the program from Example 1 is executed using the operational semantics of Definition 8.

$$\begin{aligned} & \langle mset([a]), item(b); \top; \emptyset \rangle \\ &= \langle mset(L), item(A); L = [a] \wedge A = b; \emptyset \rangle \\ &\mapsto \langle mset([A|L]); L = [a] \wedge A = b; \emptyset \rangle \\ &= \langle mset([a, b]); \top; \emptyset \rangle \end{aligned}$$

In the first step, an equivalent state with fresh local variables L and A is constructed. The constraints in the goal store of this state are syntactically equivalent to the head of the variant of the rule with variables L and A . The guard of the rule is \top and therefore, the rule is applicable. After applying the rule, the state can be transformed into a more readable form without local variables.

An important analysis technique is the merging of states.

Definition 9 (merge operator \diamond). *Let $\sigma_i = \langle \mathbb{G}_i; \mathbb{B}_i; \mathbb{V}_i \rangle$ for $i = 1, 2$ be two CHR states such that local variables of one state are disjoint from all variables in the other state. Then for a set \mathbb{V} of variables*

$$\sigma_1 \diamond_{\mathbb{V}} \sigma_2 := \langle \mathbb{G}_1 \uplus \mathbb{G}_2; \mathbb{B}_1 \wedge \mathbb{B}_2; (\mathbb{V}_1 \cup \mathbb{V}_2) \setminus \mathbb{V} \rangle.$$

For equivalence classes of CHR states, the merging is defined as $[\sigma_1] \diamond_{\mathbb{V}} [\sigma_2] := [\sigma_1 \diamond_{\mathbb{V}} \sigma_2]$ for two representatives of the equivalence class that have disjoint variables. For $\mathbb{V} = \emptyset$ we write $[\sigma_1] \diamond [\sigma_2]$ [7, p. 50, Definition 10.1].

Since local variables have to be disjoint when merging two states, it is not possible to extract information about them directly. For instance, $[\langle c(X); X = 1; \emptyset \rangle]$ is the version of $[\langle c(X); \top; \emptyset \rangle]$ with the local variable X , where X is bound to the number 1. In the state $[\langle c(X), X = 1, \emptyset \rangle]$, we would consider $X = 1$ as contextual

information about the local variable X . It is not possible to extract this information by $[\langle c(X); \top; \emptyset \rangle] \diamond [\langle \emptyset; X = 1; \emptyset \rangle]$, since $[\langle \emptyset; X = 1; \emptyset \rangle] = [\langle \emptyset; \top; \emptyset \rangle] = [\sigma_\emptyset]$, i.e. the empty state. Hence, the result of merging the two states is $[\langle c(X); \top; \emptyset \rangle]$ although we would like to see the result $[\langle c(X); X = 1; \emptyset \rangle]$.

It is necessary to rather make X a global variable first that is reduced by the merge operator $\diamond_{\{X\}}$:

$$[\langle c(X); \top; \{X\} \rangle] \diamond_{\{X\}} [\langle \emptyset; X = 1; \{X\} \rangle] = [\langle c(X); X = 1; \emptyset \rangle] = [\langle c(1); \top; \emptyset \rangle].$$

Global variables can thus be used to share information between two states that are merged. [7, p. 50, Example 10.2] In the confluence criterion in Sect. 4, we only generate states from the program source code where all variables are global.

In general, $\diamond_{\mathbb{V}}$ is not associative. However, the following lemma shows a restricted form of associativity that is used in the proof of the confluence modulo equivalence criterion in Sect. 4.

Lemma 1. *Let $\sigma_1, \sigma_2, \sigma_3$ be CHR states such that no local variable of a state occurs in another state. Then $[\sigma_1] \diamond_{\mathbb{V}} ([\sigma_2] \diamond [\sigma_3]) = ([\sigma_1] \diamond [\sigma_2]) \diamond_{\mathbb{V}} [\sigma_3]$ holds for all \mathbb{V} [7, p. 52, Lemma 10.7].*

2.3 Confluence of CHR Programs

The idea of the confluence criterion is to exploit the *monotonicity* property of CHR, i.e. that all rules applicable in one state are applicable in any larger state.

Lemma 2 (monotonicity). *If $[\sigma] \mapsto [\tau]$, then $[\sigma] \diamond_{\mathbb{V}} [\sigma'] \mapsto [\tau] \diamond_{\mathbb{V}} [\sigma']$ for all \mathbb{V} and $[\sigma]$ [7, p. 51, Lemma 10.4].*

Basic Confluence Test. Monotonicity allows us to reason from states about larger states. The idea of the basic confluence test is to construct a finite set of *rule states* that consist of the head and guard constraints of a rule and then overlap them with all other rule states. Intuitively, overlapping two rules means that a state is constructed where parts of the rule heads are equated (if possible) and the rest is just included. In such a state, both rules are applicable.

By applying the overlapping rules to the overlap state, we get a *critical pair*. Thereby, one state is the result after applying the first overlapping rule to the overlap state and the other state is the result after applying the second rule to the overlap state. If all critical pairs are joinable, the program is locally confluent. In the following, we formalize this idea. The definitions are taken from [7]. Similar definitions can be found in [1].

Definition 10 (rule state). *For a rule $r : H_k \setminus H_r \Leftrightarrow G \mid B_c, B_b$ let \mathbb{V} be the variables occurring in H_k, H_r and G . Then the state $\langle H_k \uplus H_r; G; \mathbb{V} \rangle$ is called the rule state of r . In the literature, the rule states are sometimes called minimal states. [7, p. 78, Definition 13.8].*

Definition 11 (overlap). For any two (not necessarily different) rules of a CHR program of the form $r_1 : H_k \setminus H_r \Leftrightarrow G \mid B_c, B_b, r_2 : H'_k \setminus H'_r \Leftrightarrow G' \mid B'_c, B'_b$ and with variables that are renamed apart, let $O_k \subseteq H_k, O_r \subseteq H_r, O'_k \subseteq H'_k, O'_r \subseteq H'_r$ be subsets of the heads of the rules such that for $B := ((O_k \uplus O_r) = (O'_k \uplus O'_r)) \wedge G \wedge G'$ it holds that $\mathcal{CT} \models \exists.B$ and $(O_r \uplus O'_r) \neq \emptyset$, where $\exists.B$ is the existential closure over B . Then the state $\sigma := \langle K \uplus K' \uplus R \uplus R' \uplus O_k \uplus O_r; B; \mathbb{V} \rangle$ is called an overlap of r_1 and r_2 where \mathbb{V} is the set of all variables occurring in heads and guards of both rules and $K := H_k \setminus O_k, K' := H'_k \setminus O'_k, R := H_r \setminus O_r, R' := H'_r \setminus O'_r$. The pair of states (σ_1, σ_2) with $\sigma_1 := \langle K \uplus K' \uplus R \uplus O'_k \uplus B_c; B \wedge B_b; \mathbb{V} \rangle$ and $\sigma_2 := \langle K \uplus K' \uplus R \uplus O'_k \uplus B'_c; B \wedge B'_b; \mathbb{V} \rangle$ is called critical pair of the overlap σ . The critical pair can be obtained by applying the rules to the overlap state. [7, p. 82, Definition 14.5].

Invariant-Based Confluence Test. The idea of exploiting monotonicity fails, when invariants on the states are introduced. A property \mathcal{I} is an invariant if and only if for all states $[\sigma]$ where $\mathcal{I}([\sigma])$ holds and for all $[\tau]$ with $[\sigma] \mapsto^* [\tau]$ the invariant $\mathcal{I}([\tau])$ holds as well.

If in the confluence test a constructed overlap does not satisfy the invariant, then this overlap state is not part of the transition system and therefore no information can be gained from analyzing it. It is not possible to just ignore such states as there are invariants that are not satisfied in an overlap state, but might be satisfied in a larger state. There are also invariants that are invalidated in an overlap state and that cannot be satisfied by state extension (c.f. Example 4). For instance, if only a constraint is only allowed to appear at most once in a state, this invariant cannot be satisfied by extending the state invalidating it.

Nevertheless, the idea of using overlap states for confluence analysis can be generalized, such that it can be used for invariant-based confluence. For this purpose, for an invariant \mathcal{I} and an overlap state $[\sigma]$ the set of all extensions of $[\sigma]$ such that \mathcal{I} holds – denoted by $\Sigma^{\mathcal{I}}([\sigma])$ – is considered. As this set usually is infinitely large, we want to extract a set of minimal elements of $\Sigma^{\mathcal{I}}([\sigma])$, called $\mathcal{M}^{\mathcal{I}}([\sigma])$, that have to be considered to show local confluence w.r.t. \mathcal{I} . However, for this purpose a partial order on states has to be defined. The set $\mathcal{M}^{\mathcal{I}}([\sigma])$ is finite for many invariants, but there are examples of invariants that lead to infinite sets of minimal elements.

In [5–7] the following has been proven: If we can show that for all overlap states $[\sigma]$ of a terminating program the critical pairs derived from all states in $\mathcal{M}^{\mathcal{I}}([\sigma])$ are joinable, the program is confluent w.r.t. to \mathcal{I} .

We now give formal definitions of the notions used in the above description. Since it is a commutative monoid, a partial order can be derived from the merge operator [7]:

Lemma 3 (partial order \triangleleft). For the set of CHR states Σ , the relation $\triangleleft : \Sigma \times \Sigma$ defined as $[\sigma] \triangleleft [\sigma']$ if and only if $\exists[\hat{\sigma}] . [\sigma] \diamond [\hat{\sigma}] = [\sigma']$ where $\sigma, \sigma' \in \Sigma$ is a partial order. [7, p. 53, Lemma 10.8]

In [5,6], another partial order has been defined. However, it has been shown that the relation defined there is not a partial order by mistake [7]. Therefore, we use the partial order that has first been introduced in [7, p. 53, Lemma 10.8] to avoid these problems.

For an invariant, we can now define the set of minimal elements that extend a state such that the invariant does hold.

Definition 12 (minimal elements). *For an invariant \mathcal{I} , let the set $\Sigma^{\mathcal{I}}([\sigma]) := \{[\sigma'] \mid \mathcal{I}([\sigma \diamond \sigma']) \wedge \sigma' \text{ has no local variables}\}$. The set $\mathcal{M}^{\mathcal{I}}([\sigma])$ is the set of \triangleleft -minimal elements of $\Sigma^{\mathcal{I}}([\sigma])$ such that $\forall[\sigma'] \in \Sigma^{\mathcal{I}}([\sigma]) . \exists[\sigma_m] \in \mathcal{M}^{\mathcal{I}}([\sigma]) . [\sigma_m] \triangleleft [\sigma']$ [7, p. 80, Definition 13.11]. By well-definedness of $\mathcal{M}^{\mathcal{I}}([\sigma])$ we denote that it has the latter property.*

The set $\Sigma^{\mathcal{I}}$ may become infinitely large for states with local variables. Hence in program analysis, w.l.o.g. the states are restricted to only global variables. Monotonicity of CHR (Lemma 2) ensures that all results remain applicable if any of these variables are made local [7, p. 80].

Note that for an invariant \mathcal{I} and a state $[\sigma]$ where $\mathcal{I}([\sigma])$ holds, the set of minimal extensions is $\mathcal{M}^{\mathcal{I}}([\sigma]) = \{[\sigma_\emptyset]\}$, where $\sigma_\emptyset := \langle \emptyset; \top; \emptyset \rangle$ is the empty state [7, p. 80, Lemma 13.13]. The invariant-based confluence test then coincides with the basic confluence criterion. In Sect. 4 we generalize the idea of the invariant-based confluence test for invariant-based confluence modulo equivalence.

Example 4 (Multi-Set Items (cont.)). For the multi-set program from Example 1, the following problem arises: If there is more than one *mset* constraint, the program can choose non-deterministically where to add an item. Therefore, it cannot be confluent. For instance, the following transitions in shorthand notation is possible: *mset*([a]), *mset*([b]), *item*([c]) can either end in the final state *mset*([a, c]), *mset*([b]) or *mset*([a]), *mset*([b, c]).

The problem can be solved by introducing the multi-set invariant \mathcal{S} : In every CHR state there is at most one *mset*(-) constraint. Note that the set of minimal extensions $\mathcal{M}^{\mathcal{S}}([\sigma]) = \emptyset$ for all states $[\sigma]$, as there are no extensions for states that do not satisfy the invariant (i.e. where there is more than one *mset* constraint) such that the invariant is satisfied (i.e. there is at most one *mset* constraint).

3 Compatibility of Equivalence Relations

In this section, we motivate a restriction of equivalence relations that make confluence modulo equivalence analysis manageable. Note that in the context of confluence modulo equivalence, typically user-defined equivalence relations different from state equivalence (c.f. Definition 7) are regarded. State equivalence is referred to by \equiv or by the corresponding equivalence class brackets $[\cdot]$. The symbol \approx denotes some general user-defined equivalence relation that is potentially different from \equiv (but is not required to be).

In the confluence criterion, we want to use the idea of exploiting monotonicity of CHR to reason from small states that come from the rules in the program

over all states. However, monotonicity can be broken by user-defined equivalence relations. This means that in general for two states with $[\sigma] \approx [\sigma']$, it is possible that an extension with $[\tau] \approx [\tau']$ leads to states that are not equivalent, i.e. $[\sigma] \diamond_{\vee} [\tau] \not\approx [\sigma'] \diamond_{\vee} [\tau']$ as shown in the following example.

Example 5. We construct an equivalence relation that breaks monotonicity. Let $\#c : \Sigma \rightarrow \mathbb{N}_0$ be a function that returns the number of constraints c in the goal store of a state. We separate the CHR state space Σ into two disjoint subsets:

$$\Sigma_1 := \{[\sigma] \mid \#c([\sigma]) < 3\}, \quad \Sigma_2 := \{[\sigma] \mid \#c([\sigma]) \geq 3\}.$$

The partition of the state space clearly defines an equivalence relation \approx with equivalence classes Σ_1 and Σ_2 .

Let $[\sigma_1] = [\langle c; \top; \emptyset \rangle]$ and $[\sigma_2] = [\langle c, c; \top; \emptyset \rangle]$. Since $[\sigma_1], [\sigma_2] \in \Sigma_1$, it holds that $[\sigma_1] \approx [\sigma_2]$. Let $[\tau] = [\langle c; \top; \emptyset \rangle]$. If we extend the two states by $[\tau]$, the extended states are not equivalent any more:

$$[\sigma_1] \diamond [\tau] = [\langle c, c; \top; \emptyset \rangle] \in \Sigma_1, \text{ but } [\sigma_2] \diamond [\tau] = [\langle c, c, c; \top; \emptyset \rangle] \in \Sigma_2.$$

Hence, although $[\sigma_1] \approx [\sigma_2]$, $[\sigma_1] \diamond [\tau] \not\approx [\sigma_2] \diamond [\tau]$.

This case does not harm testing for the β property, since the extended states do not have to be tested for joinability modulo equivalence according to the β property. However, we can construct the converse case: Let $[\sigma_3] = [\langle c, c, c; \top; \emptyset \rangle] \in \Sigma_2$. Then $[\sigma_2] \not\approx [\sigma_3]$. However, if the two states are extended by $[\tau]$, we get

$$[\sigma_2] \diamond [\tau] = [\langle c, c, c; \top; \emptyset \rangle] \in \Sigma_2, \text{ and } [\sigma_3] \diamond [\tau] = [\langle c, c, c, c; \top; \emptyset \rangle] \in \Sigma_2.$$

Hence, $[\sigma_2] \diamond [\tau] \approx [\sigma_3] \diamond [\tau]$, although $[\sigma_2] \not\approx [\sigma_3]$. This is critical to the β property: Now it is not possible any more to use a rule state and its equivalent states to reason about all states as we miss some larger state by this attempt.

To ensure monotonicity in the context of equivalence relations, we need equivalence to be maintained by the merge operator. The equivalence relation is then called a *congruence relation* with respect to the merge operator.

Definition 13 (congruence relation). *An equivalence relation $\approx \subseteq A \times A$ is called a congruence relation with respect to an operator $\circ : A \times A \rightarrow A$ if for all x, x', y, y' : If $x \approx x'$ and $y \approx y'$ then $x \circ y \approx x' \circ y'$.*

Unfortunately, this does not suffice to reason from rule states about any other state. It must be ensured that if two states $[\sigma]$ and $[\sigma']$ are equivalent and $[\sigma]$ can be decomposed into two parts, then $[\sigma']$ must be decomposable into two parts that are equivalent to the decomposition of $[\sigma]$. This ensures that when showing joinability of two small states, the larger states can still be joined, as they are syntactically decomposable into smaller joinable states.

Definition 14 (split property). *An equivalence relation $\approx \subseteq A \times A$ has the split property with respect to an operator $\circ : A \times A \rightarrow A$ if for all x, x_1, x_2, y : If $x = x_1 \circ x_2$ and $x \approx y$ then $\exists y_1, y_2$ such that $x_1 \approx y_1, x_2 \approx y_2$ and $y = y_1 \circ y_2$.*

The split property assumes a syntactic relation between two states that are equivalent under an equivalence relation. If a state can be split into two parts and is equivalent to another state, this state can be split into equivalent parts.

Example 6. This example defines an equivalence relation $\hat{=}$ that does not satisfy the split property. It is the smallest equivalence relation where the following two conditions hold: If $\sigma \equiv \sigma'$ then also $\sigma \hat{=} \sigma'$. Additionally, if $\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle \hat{=} \langle \mathbb{G}'; \mathbb{B}'; \mathbb{V}' \rangle$, then $\langle \{c, c\} \uplus \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle \hat{=} \langle \{d\} \uplus \mathbb{G}'; \mathbb{B}'; \mathbb{V}' \rangle$. Hence, all pairs of c constraints can be replaced by a d constraint.

The equivalence relation obviously is a congruence relation w.r.t. \diamond . However, it does not have the split property: Let $\sigma \equiv c, c$ be a CHR state in shorthand notation. Then $\sigma \equiv c \diamond c$. By definition of $\hat{=}$, we have that $\sigma \hat{=} d$. However, there are no σ_1, σ_2 such that $\sigma_1 \hat{=} c, \sigma_2 \hat{=} c$ and $d \equiv \sigma_1 \diamond \sigma_2$.

In the confluence test, for all states σ it has to be shown that if $\sigma \equiv \sigma'$ and $\sigma \mapsto_r \tau$ then $\sigma' \downarrow_{\approx} \tau$ to satisfy the β property. By the application of r to σ , we know that for the rule state σ_r, σ can be split into $[\sigma] = [\sigma_r] \diamond [\delta]$. To reason from joinability of σ_r and all its equivalent states, we also have to be able to split σ' into two parts $[\sigma'_r]$ and $[\delta']$. However, for the congruence relation $\hat{=}$ this is not possible as we have shown before. Hence, the idea of reasoning from rule states about all larger states cannot be applied.

Note that the split property is only required to hold for states where the invariant holds. Hence, by an appropriate invariant, the split property can be recovered to show confluence w.r.t. this invariant.

Definition 15 (compatibility). *An equivalence relation \approx is \circ -compatible w.r.t. an operator \circ if it is a congruence relation with the split property w.r.t. \circ .*

At first glance, compatibility is a strict property that does not seem to be satisfied by many equivalence relations. However, there are interesting \diamond -compatible equivalence relations different from the trivial state equivalence:

Example 7 (Multi-Set Items (cont.)). Example 1 is continued by introducing the following equivalence relation \approx^S that is the smallest equivalence relation on CHR states such that $[\langle \{mset(S_1)\} \uplus \mathbb{G}_1; \mathbb{B}_1; \mathbb{V}_1 \rangle] \approx^S [\langle \{mset(S_2)\} \uplus \mathbb{G}_2; \mathbb{B}_2; \mathbb{V}_2 \rangle]$ if and only if S_1 is a permutation of S_2 and $[\langle \mathbb{G}_1; \mathbb{B}_1; \mathbb{V}_1 \rangle] \approx^S [\langle \mathbb{G}_2; \mathbb{B}_2; \mathbb{V}_2 \rangle]$.

For instance, the following states in shorthand notation are equivalent according to \approx^S : $mset([a, b]), mset([c, d]), item(e) \approx^S mset([b, a]), mset([d, c]), item(e)$ and $item(a) \approx^S item(a)$. However, $mset([a, b]), item(c) \not\approx^S mset([a, b]), item(d)$ and $mset([a, b]), item(c) \not\approx^S mset([a, b]), item(c), item(c)$ because the second item c does not have a partner in the first state. Similarly, $mset([a, b]), mset([b, a]) \not\approx^S mset([a, b])$ because there is only one $mset$ constraint on the right hand side.

Note that by this definition the following holds for states with unbound variables: $[\langle mset(X); perm(X, Y); \{X, Y\} \rangle] \approx^S [\langle mset(Y); perm(X, Y); \{X, Y\} \rangle]$ where $perm(X, Y)$ is a built-in constraint that is true, if X is a permutation of Y , but $[\langle mset(X); \top; \{X\} \rangle] \not\approx^S [\langle mset(Y); \top; \{Y\} \rangle]$. The two variables X and Y are free variables and therefore it is not clear that they are permutations of each other. By adding that X is a permutation of Y , the two states are equivalent.

This equivalence relation is \diamond -compatible. For reasons of space, the proof is provided in the extended online version.¹

4 Confluence Modulo Equivalence w.r.t. an Invariant

First of all, the notion of invariant-based confluence modulo equivalence is defined.

Definition 16 (\mathcal{I} -confluence modulo \approx). *A state transition system is \mathcal{I} -confluent modulo an equivalence relation \approx for an invariant \mathcal{I} if and only if*

$$\begin{aligned} \forall \sigma_1, \sigma_2, \sigma'_1, \sigma'_2 . \mathcal{I}(\sigma_1) \wedge \mathcal{I}(\sigma'_1) \wedge \sigma_1 \approx \sigma'_1 \wedge \sigma_1 \mapsto^* \sigma_2 \wedge \sigma'_1 \mapsto^* \sigma'_2 \\ \rightarrow \exists \sigma_3, \sigma'_3 . \sigma_2 \mapsto^* \sigma_3 \wedge \sigma'_2 \mapsto^* \sigma'_3 \wedge \sigma_3 \approx \sigma'_3. \end{aligned}$$

In practice, the following restriction is made on invariants:

Definition 17 (\approx maintains \mathcal{I}). *An invariant \mathcal{I} is maintained by an equivalence relation \approx , if and only if for all states $[\sigma] \approx [\sigma']$ it holds that $\mathcal{I}([\sigma]) \leftrightarrow \mathcal{I}([\sigma'])$.*

This restriction ensures the practicability of Definition 16, since it may be inelegant and misleading if in a program that is \mathcal{I} -confluent modulo \approx there exist two equivalent states where one is part of the program (i.e. the invariant holds) and the other is not. This may have undesired effects in further analysis. Hence, the invariant and equivalence relation should be chosen such that they are compliant anyway, although it is not required by Definition 16.

The following lemma is an important generalization of the joinability corollary in [7, p. 85, Corollary 14.9] that is a direct consequence of monotonicity. The idea was that if two states are joinable, they are still joinable if they are extended by the identical state. In the context of confluence modulo equivalence, we have to generalize this approach of exploiting monotonicity such that the state extensions are not required to be syntactically identical, but equivalent for some user-defined compatible equivalence relation.

Lemma 4 (joinability). *Let \approx be a congruence relation with respect to \diamond and $[\sigma_1], [\sigma_2], [\sigma'_1], [\sigma'_2]$ be CHR states with $[\sigma'_1] \approx [\sigma'_2]$. If $[\sigma_1] \downarrow \approx [\sigma_2]$ then $([\sigma_1] \diamond_{\mathbb{V}} [\sigma'_1]) \downarrow \approx ([\sigma_2] \diamond_{\mathbb{V}} [\sigma'_2])$ for all \mathbb{V} .*

Proof. Let $[\sigma_1], [\sigma_2], [\sigma'_1], [\sigma'_2]$ be CHR states with $[\sigma'_1] \approx [\sigma'_2]$ and $[\sigma_1] \downarrow \approx [\sigma_2]$. Hence, there are CHR states $[\tau], [\tau']$ with $[\tau] \approx [\tau']$ and $[\sigma_1] \mapsto^* [\tau]$ and $[\sigma_2] \mapsto^* [\tau']$. Due to monotonicity (c.f. Lemma 2), we have that $([\sigma_1] \diamond_{\mathbb{V}} [\sigma'_1]) \mapsto^* ([\tau] \diamond_{\mathbb{V}} [\sigma'_1])$ and $([\sigma_2] \diamond_{\mathbb{V}} [\sigma'_2]) \mapsto^* ([\tau'] \diamond_{\mathbb{V}} [\sigma'_2])$. Since $[\sigma'_1] \approx [\sigma'_2]$, $[\tau] \approx [\tau']$ and \approx is a congruence relation with respect to \diamond , we have that $([\tau] \diamond_{\mathbb{V}} [\sigma'_1]) \approx ([\tau'] \diamond_{\mathbb{V}} [\sigma'_2])$.

¹ <https://arxiv.org/abs/1802.03381>.

In the next step, we provide a test for the α property in the context of an invariant. The basic idea is that we gather all overlap states and extend them with a minimal extension such that the invariant does hold. For all those minimal extensions of all overlap states we have to show joinability modulo equivalence. Formally, this leads to the following lemma.

Lemma 5 (α property test). *Let \mathcal{P} be a CHR program, \mathcal{I} an invariant, \approx a congruence relation and let $\mathcal{M}^{\mathcal{I}}([\sigma])$ be well-defined for all overlaps σ of rules in \mathcal{P} , then: \mathcal{P} has the α property with respect to \mathcal{I} and \approx if and only if for all overlaps σ with critical pairs (σ_1, σ_2) and all $[\sigma_m] \in \mathcal{M}^{\mathcal{I}}([\sigma])$ holds $([\sigma_1] \diamond [\sigma_m] \downarrow \approx [\sigma_2] \diamond [\sigma_m])$.*

Proof. For reasons of space, the proof is provided in the extended online version.² It is similar to the proof for the β property.

To prove local confluence modulo equivalence, we also have to prove the β property, i.e. we have to consider that if in a state a CHR transition is possible and the state is equivalent to another state, then the successor state and the equivalent state have to be joinable modulo equivalence. In the following lemma, we adapt the test for the α property to cover the β property.

The main idea is to reason from rule states, i.e. the head and guard constraints of rules, over all states. For this purpose, all rule states have to be extended by a minimal extension such that the invariant holds. Then all states that are equivalent to these extended rule states have to be shown to be joinable to the extended rule state after the rule has been applied. Unfortunately – depending on the invariant – in general there can be infinitely many such equivalent states. However, the idea still simplifies the proof procedure for the β property, as only rule states have to be considered in contrast to all states of the transition system.

This is not possible for general equivalence relations, but only for those that are compatible to the merge operator and that maintain the invariant.

Lemma 6 (β property test). *Let \mathcal{P} be a CHR program, \mathcal{I} an invariant, \approx a \diamond -compatible equivalence relation that maintains \mathcal{I} and let $\mathcal{M}^{\mathcal{I}}([\sigma])$ be well-defined for all rule states $[\sigma]$ in \mathcal{P} , then: \mathcal{P} has the β property with respect to \mathcal{I} and \approx if and only if for all rule states $[\sigma]$ with successor state $[\sigma_1]$, all $[\sigma_2]$ with $[\sigma] \approx [\sigma_2]$ and all $[\sigma_m^1] \in \mathcal{M}^{\mathcal{I}}([\sigma])$ and all $[\sigma_m^2] \approx [\sigma_m^1]$ where $\mathcal{I}([\sigma_2] \diamond [\sigma_m^2])$ is satisfied, it holds that $([\sigma_1] \diamond [\sigma_m^1]) \downarrow \approx ([\sigma_2] \diamond [\sigma_m^2])$.*

Proof. “ \Rightarrow ”: This follows from Definition 4 and Lemma 4.

“ \Leftarrow ”: Let $[\sigma]$, $[\sigma_1]$ and $[\sigma_2]$ be CHR states where $\mathcal{I}([\sigma])$ and $\mathcal{I}([\sigma_2])$ hold and $[\sigma] \mapsto_r [\sigma_1]$ for some rule r and $[\sigma] \approx [\sigma_2]$. Since a rule is applicable in $[\sigma]$, there is a rule state $\sigma_r = \langle _ ; _ ; \mathbb{V} \rangle$ of rule r such that for some $[\delta_1] := \langle \langle \mathbb{G}; \mathbb{B}; \mathbb{V}' \rangle \rangle$ it holds that $[\sigma] = [\sigma_r] \diamond_{\mathbb{V}} [\delta_1]$. The variables \mathbb{V} from the rule r are not part of $[\sigma]$ and are therefore removed by the merging $\diamond_{\mathbb{V}}$.

² <https://arxiv.org/abs/1802.03381>.

By definition of the rule state (c.f. Definition 10) and definition of the state transition system (c.f. Definition 8), we also have that there is a state $[\sigma'_1]$ such that $[\sigma_r] \mapsto_r [\sigma'_1]$. Due to monotonicity (c.f. Lemma 2) it holds that $[\sigma_1] = [\sigma'_1] \diamond_{\vee} [\delta_1]$.

Let $[\sigma_2] = [\sigma'_2] \diamond_{\vee} [\delta_2]$ be a partition of $[\sigma_2]$ such that $[\sigma'_2] \approx [\sigma_r]$ and $[\delta_2] \approx [\delta_1]$. Such a partition exists since \approx is \diamond -compatible and $[\sigma] \approx [\sigma_2]$ by precondition.

As $\mathcal{I}([\sigma])$ holds and w.l.o.g. $[\sigma]$ has no local variables (see the comment after Definition 12): $[\delta_1] \in \Sigma^{\mathcal{X}}([\sigma_r])$ and therefore $\exists[\sigma_m^1] \in \mathcal{M}^{\mathcal{X}}([\sigma_r]).[\sigma_m^1] \triangleleft [\delta_1]$. This means that there is a minimal element $[\sigma_m^1]$ in the set of extensions of the rule state $[\sigma_r]$ that extend $[\sigma_r]$ such that the invariant holds.

It follows by definition of \triangleleft that $\exists[\delta'_1].[\delta_1] = [\sigma_m^1] \diamond [\delta'_1]$ and hence $[\sigma] = [\sigma_r] \diamond_{\vee} ([\sigma_m] \diamond [\delta'_1])$. By Lemma 1, we get $[\sigma] = ([\sigma_r] \diamond [\sigma_m]) \diamond_{\vee} [\delta'_1]$. Analogously, by substitution of $[\delta_1]$ in $[\sigma_1]$ and due to the split property of \approx , we find that $[\sigma_i] = ([\sigma'_i] \diamond [\sigma_m^i]) \diamond_{\vee} [\delta'_i]$ for $i = 1, 2$ where $[\sigma_m^1] \approx [\sigma_m^2]$.

Since \mathcal{I} is maintained by \approx , we have by precondition that $([\sigma'_1] \diamond [\sigma_m^1]) \downarrow \approx ([\sigma'_2] \diamond [\sigma_m^2])$. Since $[\sigma_1] = ([\sigma'_1] \diamond [\sigma_m^1]) \diamond_{\vee} [\delta'_1]$ and $[\sigma_2] = ([\sigma'_2] \diamond [\sigma_m^2]) \diamond_{\vee} [\delta'_2]$ and $[\delta'_1] \approx [\delta'_2]$, we have by Lemma 4 also that $([\sigma_1] \downarrow \approx [\sigma_2])$.

Theorem 2 (confluence modulo \approx w.r.t. invariant). *Let \mathcal{I} be an invariant and \mathcal{P} an \mathcal{I} -terminating CHR program. \mathcal{P} has the α and β property with respect to \mathcal{I} and an equivalence relation \approx if and only if \mathcal{P} is \mathcal{I} -confluent modulo \approx .*

Proof. Theorem 1 is used on the reduced state transition system that only contains states where the invariant holds.

Note that for testing the α property, the criterion only assumes a congruence relation, whereas for proving the β property the split property must hold as well and the invariant must maintain equivalence.

Example 8 (Item Sets (cont.)). It is shown that the program from Example 1 is \mathcal{S} -confluent modulo $\approx^{\mathcal{S}}$.

α property. The only overlap that satisfies the invariant \mathcal{S} has the shorthand notation $item(A), item(B), mset(L)$ with critical pair $item(B), mset([A|L])$ and $item(A), mset([B|L])$. It can be reduced to $mset([B, A|L]) \approx^{\mathcal{S}} mset([A, B|L])$.

β property. All equivalences to the rule state have the form $[\langle item(A), mset(L); \top; \{A, L\} \rangle] \approx^{\mathcal{S}} [\langle item(A), mset(L'); \top; \{A, L'\} \rangle]$ where L' is a permutation of L . The preconditions of Lemma 6 are satisfied, since both states satisfy the invariant. Both states reduce to the goal stores $mset([A|L])$ and $mset([A|L'])$. It is clear that those two final states are equivalent and therefore joinable modulo $\approx^{\mathcal{S}}$. \square

5 Discussion and Related Work

The α property test is decidable for terminating programs as long as the invariant and the equivalence relation (a congruence relation w.r.t. \diamond) are decidable and

the set of minimal extensions is finite. In the β property test, the class of states that are equivalent to the rule state may be infinitely large in general.

In the multi-set example (c.f. Example 8), it can be seen that only one other state has to be considered to show joinability of all states equivalent to the rule state, since the CHR semantics allows for logical variables. In general, there might be more complicated equivalence relations that are more difficult to test.

Confluence modulo equivalence with invariants has been studied for a variant of CHR that includes non-logical built-in constraints [8,9]. This approach introduces a meta language for CHR to prove confluence modulo equivalence. It is claimed that the traditional proof methods for confluence in CHR expressed in first-order logic are not sufficient in the context of confluence modulo equivalence, especially with non-logical built-in constraints. The meta-level is claimed to allow proving confluence modulo equivalence for all equivalence relations. It is shown to be useful for programs with non-logical built-in constraints.

In many cases the analysis of programs with purely logical CHR is desired and invariants and equivalence relations behave in a way that allow for a more direct treatment. In our approach, no meta-level is necessary. It is directly available for the de facto standard of CHR semantics. It seems to us that the example in [9] indicates that for proving confluence modulo equivalence with the meta-level approach, monotonicity and therefore \diamond -compatibility are used implicitly.

Invariant-based confluence (or observable confluence) for CHR without user-defined equivalence relations has been studied in [5,6]. In [7], it has been shown that the proposed partial order is not well-defined. Our approach integrates the corrected version of invariant-based confluence as found in [7]. Additionally, it extends the idea by confluence modulo user-defined equivalence relations.

6 Conclusion and Future Work

A sufficient and necessary criterion for confluence modulo equivalence w.r.t. an invariant has been presented and formally proven (c.f. Lemmas 4 to 6 and Theorem 2). For this purpose, the set of *compatible* equivalence relations (c.f. Definitions 13 to 15) has been identified to behave well with this confluence criterion for CHR. When an equivalence relation has been shown to be compatible and maintains the invariant, it can be used directly for any program. In practice, it seems to be desirable that the equivalence relation maintains the invariant.

The approach is directly applicable for a non-trivial example (c.f. Example 8). It has been tested for other examples which indicates that the defined class of equivalence relations is actually meaningful. Decidability of the α property is maintained. For some invariants, the set of minimal extensions can be infinitely large and decidability is lost. Although the β property leads to an infinite number of states that have to be considered in general, the proofs are simplified tremendously, as only states equivalent to the finite number of rule states have to be considered.

In many cases it may suffice to only use an equivalence relation without an invariant. The problems originating from invariants are inexistent for those cases and our approach yields a sufficient and necessary criterion for confluence modulo equivalence without invariants.

For the future, we want to investigate how our approach can be unified with the meta-level approach [9] or other proof methods in the context of confluence such as case splitting. Furthermore, it could be interesting how non-confluent programs can be completed such that they become confluent modulo equivalence.

Acknowledgements. The authors would like to thank Henning Christiansen and Maja H. Kirkeby for the valuable discussions and ideas for future work. Additionally, the authors would like to thank the anonymous reviewers for their valuable comments.

References

1. Frühwirth, T.: *Constraint Handling Rules*. Cambridge University Press, New York (2009)
2. Abdennadher, S., Frühwirth, T., Meuss, H.: On confluence of Constraint Handling Rules. In: Freuder, E.C. (ed.) CP 1996. LNCS, vol. 1118, pp. 1–15. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61551-2_62
3. Abdennadher, S.: Operational semantics and confluence of constraint propagation rules. In: Smolka, G. (ed.) CP 1997. LNCS, vol. 1330, pp. 252–266. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0017444>
4. Abdennadher, S., Frühwirth, T., Meuss, H.: Confluence and semantics of constraint simplification rules. *Constraints* 4(2), 133–165 (1999)
5. Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable confluence for Constraint Handling Rules. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 224–239. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74610-2_16
6. Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable confluence for Constraint Handling Rules. In: Schrijvers, T., Frühwirth, T. (eds.) CHR 2006. K.U. Leuven, Department of Computer Science, Technical report CW 452, pp. 61–76, July 2006
7. Raiser, F.: Graph transformation systems in constraint handling rules: improved methods for program analysis. Ph.D. thesis, Ulm University, Germany (2010)
8. Christiansen, H., Kirkeby, M.H.: Confluence modulo equivalence in Constraint Handling Rules. In: Proietti, M., Seki, H. (eds.) LOPSTR 2014. LNCS, vol. 8981, pp. 41–58. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17822-6_3
9. Christiansen, H., Kirkeby, M.H.: On proving confluence modulo equivalence for Constraint Handling Rules. *Formal Aspects Comput.* 29(1), 57–95 (2017)
10. Raiser, F., Betz, H., Frühwirth, T.: Equivalence of CHR states revisited. In: Raiser, F., Sneyers, J. (eds.) 6th International Workshop on Constraint Handling Rules (CHR), KULCW, Technical report CW 555, pp. 33–48, July 2009
11. Huet, G.: Confluent reductions: abstract properties and applications to term rewriting systems. *J. ACM (JACM)* 27(4), 797–821 (1980)
12. Ohlebusch, E.: Church-Rosser theorems for abstract reduction modulo an equivalence relation. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 17–31. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0052358>



On Probabilistic Term Rewriting

Martin Avanzini^{1(✉)}, Ugo Dal Lago^{1,2}, and Akihisa Yamada³

¹ Inria Sophia Antipolis, Valbonne, France
`martin.avanzini@inria.fr`

² Department of Computer Science, University of Bologna, Bologna, Italy

³ National Institute of Informatics, Tokyo, Japan

Abstract. We study the termination problem for probabilistic term rewrite systems. We prove that the interpretation method is sound and complete for a strengthening of positive almost sure termination, when abstract reduction systems and term rewrite systems are considered. Two instances of the interpretation method—polynomial and matrix interpretations—are analyzed and shown to capture interesting and non-trivial examples when automated. We capture probabilistic computation in a novel way by means of multidistribution reduction sequences, thus accounting for both the nondeterminism in the choice of the redex and the probabilism intrinsic in firing each rule.

1 Introduction

Interactions between computer science and probability theory are pervasive and extremely useful to the first discipline. Probability theory indeed offers models that enable *abstraction*, but it also suggests a new model of *computation*, like in randomized computation or cryptography [17]. All this has stimulated the study of probabilistic computational models and programming languages: probabilistic variations on well-known models like automata [24], Turing machines [26], and the λ -calculus [25] are known from the early days of theoretical computer science.

The simplest way probabilistic choice can be made available in programming is endowing the language of programs with an operator modeling sampling from (one or many) distributions. Fair, binary, probabilistic choice is for example perfectly sufficient to get universality if the underlying programming language is itself universal (e.g., see [10]).

Term rewriting [27] is a well-studied model of computation when no probabilistic behavior is involved. It provides a faithful model of pure functional programming which is, up to a certain extent, also adequate for modeling higher-order parameter passing [12]. What is peculiar in term rewriting is that, in principle, rule selection turns reduction into a potentially nondeterministic process. The following question is then a natural one: is there a way to generalize term rewriting to a fully-fledged *probabilistic* model of computation? Actually, not much is known about probabilistic term rewriting: we are only aware of the definitions due to Agha et al. [1] and due to Bournez and Garnier [5]. We base our work on the latter, where probabilistic rewriting is captured as a Markov

decision process; rule selection remains nondeterministic, but each rule can have one of many possible outcomes, each with its own probability. Rewriting thus becomes a process in which both nondeterministic and probabilistic aspects are present and intermingled. When firing a rule, the reduction process implicitly samples from a distribution, much in the same way as when performing binary probabilistic choice in one of the models mentioned above.

In this paper, we first define a new, simple framework for discrete probabilistic reduction systems, which properly generalizes standard abstract reduction systems [27]. In particular, what plays the role of a reduction sequence, usually a (possibly infinite) sequence $a_1 \rightarrow a_2 \rightarrow \dots$ of *states*, is a sequence $\mu_1 \rightsquigarrow \mu_2 \rightsquigarrow \dots$ of (*multi*)*distributions* over the set of states. A multidistribution is not merely a distribution, and this is crucial to appropriately account for both the probabilistic behaviour of each rule and the nondeterminism in rule selection. Such correspondence does not exist in Bournez and Garnier’s framework, as nondeterminism has to be resolved by a *strategy*, in order to define reduction sequences. However, the two frameworks turn out to be equiexpressive, at least as far as every rule has finitely many possible outcomes. We then prove that the probabilistic ranking functions [5] are sound and complete for proving *strong almost sure termination*, a strengthening of *positive almost sure termination* [5]. We moreover show that ranking functions provide bounds on expected runtimes.

This paper’s main contribution, then, is the definition of a simple framework for *probabilistic term rewrite systems* as an example of this abstract framework. Our main aim is studying whether any of the well-known techniques for termination of term rewrite systems can be generalized to the probabilistic setting, and whether they can be automated. We give positive answers to these two questions, by describing how polynomial and matrix interpretations can indeed be turned into instances of probabilistic ranking functions, thus generalizing them to the more general context of probabilistic term rewriting. We moreover implement these new techniques into the termination tool NaTT [28]. The implementation and an extended version of this paper [3] are available at <http://www.trs.cm.is.nagoya-u.ac.jp/NaTT/probabilistic>.

2 Related Work

Termination is a crucial property of programs, and has been widely studied in term rewriting. Tools checking and certifying termination of term rewrite systems are nowadays capable of implementing tens of different techniques, and can prove termination of a wide class of term rewrite systems, although the underlying verification problem is well known to be undecidable [27].

Termination remains an interesting and desirable property in a probabilistic setting, e.g., in probabilistic programming [18] where inference algorithms often rely on the underlying program to terminate. But what does termination *mean* when systems become probabilistic? If one wants to stick to a *qualitative* definition, almost-sure termination is a well-known answer: a probabilistic computation is said to almost surely terminate iff non-termination occurs with null

probability. One could even require *positive* almost-sure termination, which asks the expected time to termination to be finite. Recursion-theoretically, checking (positive) almost-sure termination is harder than checking termination of non-probabilistic programs, where termination is at least recursively enumerable, although undecidable: in a universal probabilistic imperative programming language, almost sure termination is Π_2^0 complete, while positive almost-sure termination is Σ_2^0 complete [20].

Many sound verification methodologies for probabilistic termination have recently been introduced (see, e.g., [5, 6, 9, 14, 16]). In particular, the use of ranking martingales has turned out to be quite successful when the analyzed program is imperative, and thus does not have an intricate recursive structure. When the latter holds, techniques akin to sized types have been shown to be applicable [11]. Finally, as already mentioned, the current work can be seen as stemming from the work by Bournez et al. [5–7]. The added value compared to their work are first of all the notion of multidistribution as a way to give an instantaneous description of the state of the underlying system which exhibits both nondeterministic and probabilistic features. Moreover, an interpretation method inspired by ranking functions is made more general here, this way accommodating not only interpretations over the real numbers, but also interpretations over vectors, in the sense of matrix interpretations. Finally, we provide an automation of polynomial and matrix interpretation inference here, whereas nothing about implementation was presented in Bournez and Garnier’s work.

3 Probabilistic Abstract Reduction Systems

An *abstract reduction system* (ARS) on a set A is a binary relation $\rightarrow \subseteq A \times A$. Having $a \rightarrow b$ means that a reduces to b in one step, or b is a one-step reduct of a . Bournez and Garnier [5] extended the ARS formalism to probabilistic computations, which we will present here using slightly different notations.

We write $\mathbb{R}_{\geq 0}$ for the set of non-negative reals. A (*probability*) *distribution* on a countable set A is a function $d : A \rightarrow \mathbb{R}_{\geq 0}$ such that $\sum_{a \in A} d(a) = 1$. We say a distribution d is *finite* if its *support* $\text{Supp}(d) := \{a \in A \mid d(a) > 0\}$ is finite, and write $\{d(a_1) : a_1, \dots, d(a_n) : a_n\}$ for d if $\text{Supp}(d) = \{a_1, \dots, a_n\}$ (with pairwise distinct a_i s). We write $\text{FDist}(A)$ for the set of finite distributions on A .

Definition 1 (PARS, [5]). A probabilistic reduction over a set A is a pair of $a \in A$ and $d \in \text{FDist}(A)$, written $a \rightarrow d$. A probabilistic ARS (PARS) \mathcal{A} over A is a (typically infinite) set of probabilistic reductions. An object $a \in A$ is called *terminal* (or a normal form) in \mathcal{A} , if there is no d with $a \rightarrow d \in \mathcal{A}$. With $\text{TRM}(\mathcal{A})$ we denote the set of terminals in \mathcal{A} .

The intended meaning of $a \rightarrow d \in \mathcal{A}$ is that “there is a reduction step $a \rightarrow_{\mathcal{A}} b$ with probability $d(b)$ ”.

Example 2 (Random walk). A random walk over \mathbb{N} with *bias* probability p is modeled by the PARS \mathcal{W}_p consisting of the probabilistic reduction

$$n + 1 \rightarrow \{p : n, 1 - p : n + 2\} \quad \text{for all } n \in \mathbb{N}.$$

A PARS describes both nondeterministic and probabilistic choice; we say a PARS \mathcal{A} is *nondeterministic* if $a \rightarrow d_1, a \rightarrow d_2 \in \mathcal{A}$ with $d_1 \neq d_2$. In this case, the distribution of one-step reducts of a is nondeterministically chosen from d_1 and d_2 . Bournez and Garnier [5] describe reduction sequences via stochastic sequences, which demand nondeterminism to be resolved by fixing a *strategy* (also called *policies*). In contrast, we capture nondeterminism by defining a reduction relation $\rightsquigarrow_{\mathcal{A}}$ on distributions, and emulate ARSs by $\{1 : a\} \rightsquigarrow_{\mathcal{A}} \{1 : b\}$ when $a \rightarrow \{1 : b\} \in \mathcal{A}$. For the probabilistic case, taking Example 2 we would like to have

$$\{1 : 1\} \rightsquigarrow_{\mathcal{W}_{\frac{1}{2}}} \left\{ \frac{1}{2} : 0, \frac{1}{2} : 2 \right\},$$

meaning that the distribution of one-step reducts of 1 is $\{\frac{1}{2} : 0, \frac{1}{2} : 2\}$. Continuing the reduction, what should the distribution of two-step reducts of 1 be? Actually, it cannot be a distribution (on A): by probability $\frac{1}{2}$ we have no two-step reduct of 1. One solution, taken by [5], is to introduce $\perp \notin A$ representing the case where no reduct exists. We take another solution: we consider *subdistributions*, i.e. generalizations of distributions where probabilities may sum up to less than one, allowing

$$\{1 : 1\} \rightsquigarrow_{\mathcal{W}_{\frac{1}{2}}} \left\{ \frac{1}{2} : 0, \frac{1}{2} : 2 \right\} \rightsquigarrow_{\mathcal{W}_{\frac{1}{2}}} \left\{ \frac{1}{4} : 1, \frac{1}{4} : 3 \right\}.$$

Further continuing the reduction, one would expect $\{\frac{1}{8} : 0, \frac{1}{4} : 2, \frac{1}{8} : 4\}$ as the next step, but note that a half of the probability $\frac{1}{4}$ of 2 is the probability of reduction sequence $2 \rightarrow_{\mathcal{W}_{\frac{1}{2}}} 1 \rightarrow_{\mathcal{W}_{\frac{1}{2}}} 2$, and the other half is of $2 \rightarrow_{\mathcal{W}_{\frac{1}{2}}} 3 \rightarrow_{\mathcal{W}_{\frac{1}{2}}} 2$.

Example 3. Consider the PARS \mathcal{N} consisting of the following rules:

$$\begin{array}{lll} \mathbf{a} \rightarrow \left\{ \frac{1}{2} : \mathbf{b}_1, \frac{1}{2} : \mathbf{b}_2 \right\} & \mathbf{b}_1 \rightarrow \{1 : \mathbf{c}\} & \mathbf{c} \rightarrow \{1 : \mathbf{d}_1\} \\ & \mathbf{b}_2 \rightarrow \{1 : \mathbf{c}\} & \mathbf{c} \rightarrow \{1 : \mathbf{d}_2\}. \end{array}$$

Reducing \mathbf{a} twice always yields \mathbf{c} , so the distribution of the two-step reduct of \mathbf{a} is $\{1 : \mathbf{c}\}$. More precisely, there are two paths to reach \mathbf{c} : $\mathbf{a} \rightarrow_{\mathcal{N}} \mathbf{b}_1 \rightarrow_{\mathcal{N}} \mathbf{c}$ and $\mathbf{a} \rightarrow_{\mathcal{N}} \mathbf{b}_2 \rightarrow_{\mathcal{N}} \mathbf{c}$. Each of them can be nondeterministically continued to \mathbf{d}_1 and \mathbf{d}_2 , so the distribution of three-step reducts of \mathbf{a} is the nondeterministic choice among $\{1 : \mathbf{d}_1\}, \{\frac{1}{2} : \mathbf{d}_1, \frac{1}{2} : \mathbf{d}_2\}, \{1 : \mathbf{d}_2\}$. On the other hand, whereas it is obvious that the two-step reduct $\{1 : \mathbf{c}\}$ of \mathbf{a} should further reduce to $\{1 : \mathbf{d}_1\}$ or $\{1 : \mathbf{d}_2\}$, respectively, obtaining the third choice $\{\frac{1}{2} : \mathbf{d}_1, \frac{1}{2} : \mathbf{d}_2\}$ would require that the reduction relation $\rightsquigarrow_{\mathcal{N}}$ is defined in a non-local manner.

To overcome this problem, we refine distributions to multidistributions.

Definition 4 (Multidistributions). A multidistribution on A is a finite multiset μ of pairs of $a \in A$ and $0 \leq p \leq 1$, written $p : a$, such that

$$|\mu| := \sum_{p:a \in \mu} p \leq 1.$$

We denote the set of multidistributions on A by $\text{FMDist}(A)$.

Abusing notation, we identify $\{p_1 : a_1, \dots, p_n : a_n\} \in \text{FDist}(A)$ with multidistribution $\{\{p_1 : a_1, \dots, p_n : a_n\}\}$ as no confusion can arise. For a function $f : A \rightarrow B$, we often generalize the domain and range to multidistributions as follows:

$$f(\{p_1 : a_1, \dots, p_n : a_n\}) := \{p_1 : f(a_1), \dots, p_n : f(a_n)\}.$$

The *scalar multiplication* of a multidistribution is $p \cdot \{q_1 : a_1, \dots, q_n : a_n\} := \{p \cdot q_1 : a_1, \dots, p \cdot q_n : a_n\}$, which is also a multidistribution if $0 \leq p \leq 1$. More generally, multidistributions are closed under *convex multiset unions*, defined as $\biguplus_{i=1}^n p_i \cdot \mu_i$ with $p_1, \dots, p_n \geq 0$ and $p_1 + \dots + p_n \leq 1$.

Now we introduce the reduction relation $\rightsquigarrow_{\mathcal{A}}$ over multidistributions.

Definition 5 (Probabilistic Reduction). *Given a PARS \mathcal{A} , we define the probabilistic reduction relation $\rightsquigarrow_{\mathcal{A}} \subseteq \text{FMDist}(A) \times \text{FMDist}(A)$ as follows:*

$$\frac{a \in \text{TRM}(\mathcal{A})}{\{\{1 : a\}\} \rightsquigarrow_{\mathcal{A}} \emptyset} \quad \frac{a \rightarrow d \in \mathcal{A}}{\{\{1 : a\}\} \rightsquigarrow_{\mathcal{A}} d} \quad \frac{\mu_1 \rightsquigarrow_{\mathcal{A}} \rho_1 \quad \dots \quad \mu_n \rightsquigarrow_{\mathcal{A}} \rho_n}{\biguplus_{i=1}^n p_i \cdot \mu_i \rightsquigarrow_{\mathcal{A}} \biguplus_{i=1}^n p_i \cdot \rho_i}$$

In the last rule, we assume $p_1, \dots, p_n \geq 0$ and $p_1 + \dots + p_n \leq 1$. We denote by $\mathcal{A}(\mu)$ the set of all possible reduction sequences from μ , i.e., $\{\mu_i\}_{i \in \mathbb{N}} \in \mathcal{A}(\mu)$ iff $\mu_0 = \mu$ and $\mu_i \rightsquigarrow_{\mathcal{A}} \mu_{i+1}$ for any $i \in \mathbb{N}$.

Thus $\mu \rightsquigarrow_{\mathcal{A}} \nu$ if ν is obtained from μ by replacing every nonterminal a in μ with all possible reducts with respect to some $a \rightarrow d \in \mathcal{A}$, suitably weighted by probabilities, and by removing terminals. The latter implies that $|\mu|$ is not preserved during reduction: it decreases by the probabilities of terminals.

To continue Example 2, we have the following reduction sequence:

$$\begin{aligned} \{\{1 : 1\}\} &\rightsquigarrow_{\mathcal{W}_{\frac{1}{2}}} \{\{\frac{1}{2} : 0, \frac{1}{2} : 2\}\} \rightsquigarrow_{\mathcal{W}_{\frac{1}{2}}} \emptyset \uplus \{\{\frac{1}{4} : 1, \frac{1}{4} : 3\}\} \\ &\rightsquigarrow_{\mathcal{W}_{\frac{1}{2}}} \{\{\frac{1}{8} : 0, \frac{1}{8} : 2\}\} \uplus \{\{\frac{1}{8} : 2, \frac{1}{8} : 4\}\} \rightsquigarrow_{\mathcal{W}_{\frac{1}{2}}} \dots \end{aligned}$$

The use of multidistributions resolves the issues indicated in Example 3 when dealing with nondeterministic systems. We have, besides others, the reduction

$$\{\{1 : \mathbf{a}\}\} \rightsquigarrow_{\mathcal{N}} \{\{\frac{1}{2} : \mathbf{b}_1, \frac{1}{2} : \mathbf{b}_2\}\} \rightsquigarrow_{\mathcal{N}} \{\{\frac{1}{2} : \mathbf{c}, \frac{1}{2} : \mathbf{c}\}\} \rightsquigarrow_{\mathcal{N}} \{\{\frac{1}{2} : \mathbf{d}_1, \frac{1}{2} : \mathbf{d}_2\}\}.$$

The final step is possible because $\{\{\frac{1}{2} : \mathbf{c}, \frac{1}{2} : \mathbf{c}\}\}$ is not collapsed to $\{\{1 : \mathbf{c}\}\}$.

When every probabilistic reduction in \mathcal{A} is of form $a \rightarrow \{1 : b\}$ for some b , then $\rightsquigarrow_{\mathcal{A}}$ simulates the non-probabilistic ARS via the relation $\{\{1 : \cdot\}\} \rightsquigarrow_{\mathcal{A}} \{\{1 : \cdot\}\}$. Only a little care is needed as normal forms are followed by \emptyset .

Proposition 6. *Let \hookrightarrow be an ARS and define \mathcal{A} by $a \rightarrow \{1 : b\} \in \mathcal{A}$ iff $a \hookrightarrow b$. Then $\{\{1 : a\}\} \rightsquigarrow_{\mathcal{A}} \mu$ iff either $a \hookrightarrow b$ and $\mu = \{\{1 : b\}\}$ for some b , or $\mu = \emptyset$ and a is a normal form in \hookrightarrow .*

3.1 Notions of Probabilistic Termination

A binary relation \rightarrow is called *terminating* if it does not give rise to an infinite sequence $a_1 \rightarrow a_2 \rightarrow \dots$. In a probabilistic setting, infinite sequences are problematic only if they occur with *non-null* probability.

Definition 7 (AST). A PARS \mathcal{A} is almost surely terminating (AST) if for any reduction sequence $\{\mu_i\}_{i \in \mathbb{N}} \in \mathcal{A}(\mu)$, it holds that $\lim_{n \rightarrow \infty} |\mu_n| = 0$.

Intuitively, $|\mu_n|$ is the probability of having n -step reducts, so its tendency towards zero indicates that infinite reductions occur with zero probability.

Example 8 (Example 2 Revisited). The system \mathcal{W}_p is AST for $p \leq \frac{1}{2}$, whereas it is not for $p > \frac{1}{2}$. Note that although $\mathcal{W}_{\frac{1}{2}}$ is AST, the expected number of reductions needed to reach a terminal is infinite.

The notion of *positive almost sure termination (PAST)*, due to Bournez and Garnier [5], constitutes a refinement of AST demanding that in addition, the expected length of reduction is finite for every initial state a , independent of the employed strategy. In particular, $\mathcal{W}_{\frac{1}{2}}$ is not PAST. The expected length of a derivation can be concisely expressed in our setting as follows.

Definition 9 (Expected Derivation Length). Let \mathcal{A} be a PARS and $\mu = \{\mu_i\}_{i \in \mathbb{N}} \in \mathcal{A}(\mu)$. We define the expected derivation length $\text{edl}(\mu) \in \mathbb{R} \cup \{\infty\}$ of μ by

$$\text{edl}(\mu) := \sum_{i \geq 1} |\mu_i| .$$

A PARS \mathcal{A} is called *PAST* if for every reduction μ starting from a , $\text{edl}(\mu)$ is bounded. Without fixing a strategy, however, this condition does not ensure bounds on the derivation length.

Example 10. Consider the (non-probabilistic) ARS on $\mathbb{N} \cup \{\omega\}$ with reductions $\omega \rightarrow n$ and $n + 1 \rightarrow n$ for every $n \in \mathbb{N}$. It is easy to see that every reduction sequence is of finite length, and thus, this ARS is PAST. There is, however, no global bound on the length of reduction sequences starting from ω .

Hence we introduce a stronger notion, which actually plays a more essential role than PAST. It is based on a natural extension of *derivation height* from complexity analysis of term rewriting.

Definition 11 (Strong AST). A PARS \mathcal{A} is strongly almost surely terminating (SAST) if the expected derivation height $\text{edh}_{\mathcal{A}}(a)$ of every $a \in A$ is finite, where $\text{edh}_{\mathcal{A}}(a) \in \mathbb{R} \cup \{\infty\}$ of a is defined by

$$\text{edh}_{\mathcal{A}}(a) := \sup_{\mu \in \mathcal{A}(\{1:a\})} \text{edl}(\mu) .$$

3.2 Probabilistic Ranking Functions

Bournez and Garnier [5] generalized *ranking functions*, a popular and classical method for proving termination of non-probabilistic systems, to PARS. We give here a simpler but equivalent definition of probabilistic ranking function, taking advantage of the notion of multidistribution.

For a (multi)distribution μ over real numbers, the *expected value* of μ is denoted by $\mathbb{E}(\mu) := \sum_{p:x \in \mu} p \cdot x$. A function $f : A \rightarrow \mathbb{R}$ is naturally generalized to $f : \text{FMDist}(A) \rightarrow \text{FMDist}(\mathbb{R})$, so for $\mu \in \text{FMDist}(A)$, $\mathbb{E}(f(\mu)) = \sum_{p:x \in \mu} p \cdot f(x)$. For $\epsilon > 0$ we define the order $>_\epsilon$ on \mathbb{R} by $x >_\epsilon y$ iff $x \geq \epsilon + y$.

Definition 12. *Given a PARS \mathcal{A} on A , we say that a function $f : A \rightarrow \mathbb{R}_{\geq 0}$ is a (probabilistic) ranking function (sometimes referred to as Lyapunov ranking function), if there exists $\epsilon > 0$ such that $a \rightarrow d \in \mathcal{A}$ implies $f(a) >_\epsilon \mathbb{E}(f(d))$.*

The above definition slightly differs from the formulation in [5]: the latter demands the *drift* $\mathbb{E}(f(d)) - f(a)$ to be at least $-\epsilon$, which is equivalent to $f(a) >_\epsilon \mathbb{E}(f(d))$; and allows any lower bound $\inf_{a \in A} f(a) > -\infty$, which can be easily turned into 0 by adding the lower bound to the ranking function.

We prove that a ranking function ensures SAST and gives a bound on expected derivation length. Essentially the same result can be found in [9], but we use only elementary mathematics not requiring notions from probability theory. We moreover show that this method is complete for proving SAST.

Lemma 13. *Let f be a ranking function for a PARS \mathcal{A} . Then there exists $\epsilon > 0$ such that $\mathbb{E}(f(\mu)) \geq \mathbb{E}(f(\nu)) + \epsilon \cdot |\nu|$ whenever $\mu \rightsquigarrow_{\mathcal{A}} \nu$.*

Proof. As f is a ranking function for \mathcal{A} , we have $\epsilon > 0$ such that $a \rightarrow d \in \mathcal{A}$ implies $f(a) >_\epsilon \mathbb{E}(f(d))$. Consider $\mu \rightsquigarrow_{\mathcal{A}} \nu$. We prove the claim by induction on the derivation of $\mu \rightsquigarrow_{\mathcal{A}} \nu$.

- Suppose $\mu = \{\{1 : a\}\}$ and $a \in \text{TRM}(\mathcal{A})$. Then $\nu = \emptyset$ and $\mathbb{E}(f(\mu)) \geq 0 = \mathbb{E}(f(\nu)) + \epsilon \cdot |\nu|$ since $\mathbb{E}(f(\emptyset)) = |\emptyset| = 0$.
- Suppose $\mu = \{\{1 : a\}\}$ and $a \rightarrow \nu \in \mathcal{A}$. From the assumption $\mathbb{E}(f(\mu)) = f(a) >_\epsilon \mathbb{E}(f(\nu))$, and as $|\nu| = 1$ we conclude $\mathbb{E}(f(\mu)) \geq \mathbb{E}(f(\nu)) + \epsilon \cdot |\nu|$.
- Suppose $\mu = \biguplus_{i=1}^n p_i \cdot \mu_i$, $\nu = \biguplus_{i=1}^n p_i \cdot \nu_i$, and $\mu_i \rightsquigarrow_{\mathcal{A}} \nu_i$ for all $1 \leq i \leq n$. Induction hypothesis gives $\mathbb{E}(f(\mu_i)) \geq \mathbb{E}(f(\nu_i)) + \epsilon \cdot |\nu_i|$. Thus,

$$\begin{aligned} \mathbb{E}(f(\mu)) &= \sum_{i=1}^n p_i \cdot \mathbb{E}(f(\mu_i)) \geq \sum_{i=1}^n p_i \cdot (\mathbb{E}(f(\nu_i)) + \epsilon \cdot |\nu_i|) \\ &= \sum_{i=1}^n p_i \cdot \mathbb{E}(f(\nu_i)) + \epsilon \cdot \sum_{i=1}^n p_i \cdot |\nu_i| = \mathbb{E}(f(\nu)) + \epsilon \cdot |\nu|. \quad \square \end{aligned}$$

Lemma 14. *Let f be a ranking function for PARS \mathcal{A} . Then there is $\epsilon > 0$ such that $\mathbb{E}(f(\mu_0)) \geq \epsilon \cdot \text{edl}(\boldsymbol{\mu})$ for every $\boldsymbol{\mu} = \{\mu_i\}_{i \in \mathbb{N}} \in \mathcal{A}(\mu_0)$.*

Proof. We first show $\mathbb{E}(f(\mu_m)) \geq \sum_{i=m+1}^n |\mu_i|$ for every $n \geq m$, by induction on $m - n$. Let ϵ be given by Lemma 13. The base case is trivial, so let us consider the inductive step. By Lemma 13 and induction hypothesis we get

$$\begin{aligned} \mathbb{E}(f(\mu_m)) &\geq \mathbb{E}(f(\mu_{m+1})) + \epsilon \cdot |\mu_{m+1}| \\ &\geq \epsilon \cdot \sum_{i=m+2}^n |\mu_i| + \epsilon \cdot |\mu_{m+1}| = \epsilon \cdot \sum_{i=m+1}^n |\mu_i|. \end{aligned}$$

By fixing $m = 0$, we conclude that the sequence $\{\epsilon \cdot \sum_{i=1}^n |\mu_i|\}_{n \geq 1}$ is bounded by $\mathbb{E}(f(\mu_0))$, and so is its limit $\epsilon \cdot \sum_{i \geq 1} |\mu_i| = \epsilon \cdot \text{edl}(\mu)$. \square

Theorem 15. *Ranking functions are sound and complete for proving SAST.*

Proof. For soundness, let f be a ranking function for a PARS \mathcal{A} . For every derivation μ starting from $\{\{1 : a\}\}$, we have $\text{edl}(\mu) \leq \frac{f(a)}{\epsilon}$ by Lemma 14. Hence, $\text{edh}_{\mathcal{A}}(a) \leq \frac{f(a)}{\epsilon}$, concluding that \mathcal{A} is SAST.

For completeness, suppose that \mathcal{A} is SAST, and let $a \rightarrow d \in \mathcal{A}$. Then we have $\text{edh}_{\mathcal{A}}(a) \in \mathbb{R}$, and

$$\begin{aligned} \text{edh}_{\mathcal{A}}(a) &= \sup_{\mu \in \mathcal{A}(\{\{1:a\}\})} \text{edl}(\mu) \geq \sup_{\mu \in \mathcal{A}(d)} (1 + \text{edl}(\mu)) \\ &= 1 + \sup_{\mu \in \mathcal{A}(d)} \text{edl}(\mu) = 1 + \mathbb{E}(\text{edh}_{\mathcal{A}}(d)), \end{aligned}$$

concluding $\text{edh}_{\mathcal{A}}(a) >_1 \mathbb{E}(\text{edh}_{\mathcal{A}}(d))$. Thus, taking $\epsilon = 1$, $\text{edh}_{\mathcal{A}}$ is a ranking function according to Definition 12. \square

Bournez and Garnier claimed that ranking functions are complete for proving PAST, if the system is finitely branching [5, Theorem 3]. The claim does not hold,¹ as the following example illustrates that PAST and SAST do not coincide even for finitely branching systems.²

Example 16. Consider PARS \mathcal{A} over $\mathbb{N} \cup \{a_n \mid n \in \mathbb{N}\}$, consisting of

$$a_n \rightarrow \{\tfrac{1}{2} : a_{n+1}, \tfrac{1}{2} : 0\} \quad a_n \rightarrow \{1 : 2^n \cdot n\} \quad n + 1 \rightarrow \{1 : n\}.$$

Then P is finitely branching and PAST, because every reduction sequence from $\{\{1 : a_n\}\}$ with $n \in \mathbb{N}$ is one of the following forms:

- $\mu_{n,0} = \{\{1 : a_n\}\} \rightsquigarrow \{\{1 : 2^n \cdot n\}\} \rightsquigarrow^{2^n \cdot n} \{\{1 : 0\}\}$;
- $\mu_{n,m} = \{\{1 : a_n\}\} \rightsquigarrow^m \{\{\tfrac{1}{2^m} : a_{n+m}, \tfrac{1}{2^m} : 0\}\} \rightsquigarrow \{\{\tfrac{1}{2^m} : 2^{n+m} \cdot (n+m)\}\} \rightsquigarrow^{2^{n+m} \cdot (n+m)} \{\{\tfrac{1}{2^m} : 0\}\}$ with $m = 1, 2, \dots$;
- $\mu_{n,\infty} = \{\{1 : a_n\}\} \rightsquigarrow \{\{\tfrac{1}{2} : a_{n+1}, \tfrac{1}{2} : 0\}\} \rightsquigarrow \{\{\tfrac{1}{4} : a_{n+2}, \tfrac{1}{4} : 0\}\} \rightsquigarrow \dots$,

and $\text{edl}(\mu_{n,\alpha})$ is finite for each $n \in \mathbb{N}$ and $\alpha \in \mathbb{N} \cup \{\infty\}$. However, e.g., $\text{edh}_{\mathcal{A}}(a_0)$ is not bounded, since $\text{edl}(\mu_{0,m}) = \frac{1}{2^0} + \dots + \frac{1}{2^{m-1}} + \frac{1}{2^m} + \frac{1}{2^m} \cdot (2^m \cdot m) \geq m$ for every $m \in \mathbb{N}$.

¹ The completeness claim of [5] has already been refuted in [14], but [14] also contradicts our completeness result. The counterexample there is invalid since a part of reduction steps are not counted. We thank Luis María Ferrer Fioriti for this analysis.

² We are grateful to the anonymous reviewer who pointed us to this example.

3.3 Relation to Formulation by Bournez and Garnier

As done by Bournez and Garnier [5], the dynamics of probabilistic systems are commonly defined as stochastic sequences, i.e., infinite sequences of random variables whose n -th variable represents the n -th reduct. A disadvantage of this approach is that nondeterministic choices have to be *a priori* resolved by means of strategies. In this section, we establish a precise correspondence between our formulation and the one of Bournez and Garnier. In particular, we show that the corresponding notions of AST and PAST coincide.

We shortly recap central definitions from [5]. We assume basic familiarity with stochastic processes, see e.g. [23]. Here we fix a PARS \mathcal{A} on A . A *history* (of length $n + 1$) is a finite sequence $\mathbf{a} = a_0, a_1, \dots, a_n$ of objects from A , and such a sequence is called *terminal* if a_n is. A *strategy* ϕ is a function from nonterminal histories to distributions such that $a_n \rightarrow \phi(a_0, a_1, \dots, a_n) \in \mathcal{A}$. A history a_0, a_1, \dots, a_n is called *realizable under ϕ* iff for every $0 \leq i < n$, it holds that $\phi(a_0, a_1, \dots, a_i)(a_{i+1}) > 0$.

Definition 17 (Stochastic Reduction, [5]). *Let \mathcal{A} be a PARS on A and $\perp \notin A$ a special symbol. A sequence of random variables $\mathbf{X} = \{X_n\}_{n \in \mathbb{N}}$ over $A \cup \{\perp\}$ is a (stochastic) reduction in \mathcal{A} (under strategy ϕ) if*

$$\begin{aligned} \mathbb{P}(X_{n+1} = \perp \mid X_n = \perp) &= 1; \\ \mathbb{P}(X_{n+1} = \perp \mid X_n = a) &= 1 && \text{if } a \text{ is terminal}; \\ \mathbb{P}(X_{n+1} = \perp \mid X_n = a) &= 0 && \text{if } a \text{ is nonterminal}; \\ \mathbb{P}(X_{n+1} = a \mid X_n = a_n, \dots, X_0 = a_0) &= d(a) && \text{if } \phi(a_0, \dots, a_n) = d, \end{aligned}$$

where a_0, \dots, a_n is a realizable nonterminal history under ϕ .

Thus, \mathbf{X} is set up so that trajectories correspond to reductions $a_0 \rightarrow_{\mathcal{A}} a_1 \rightarrow_{\mathcal{A}} \dots$, and \perp signals termination. In correspondence, the derivation length is given by the *first hitting time* to \perp :

Definition 18 ((P)AST of [5]). *For $\mathbf{X} = \{X_n\}_{n \in \mathbb{N}}$ define the random variable $T_{\mathbf{X}} := \min\{n \in \mathbb{N} \mid X_n = \perp\}$, where $\min \emptyset = \infty$ by convention. A PARS \mathcal{A} is stochastically AST (resp. PAST) if for every stochastic reduction \mathbf{X} in \mathcal{A} , $\mathbb{P}(T_{\mathbf{X}} = \infty) = 0$ (resp. $\mathbb{E}(T_{\mathbf{X}}) < \infty$).*

A proof of the following correspondence is available in the extended version [3].

Lemma 19. *For each stochastic reduction $\{X_n\}_{n \in \mathbb{N}}$ in a PARS \mathcal{A} there exists a corresponding reduction sequence $\mu_0 \rightsquigarrow_{\mathcal{A}} \mu_1 \rightsquigarrow_{\mathcal{A}} \dots$ where μ_0 is a distribution and $\mathbb{P}(X_n = a) = \sum_{p:a \in \mu_n} p$ for all $n \in \mathbb{N}$ and $a \in A$, and vice versa.*

As the above lemma relates $T_{\mathbf{X}}$ with the n -th reduction μ_n of the corresponding reduction so that $\mathbb{P}(T_{\mathbf{X}} \geq n) = \mathbb{P}(X_n \neq \perp) = |\mu_n|$, using that $\mathbb{E}(T_{\mathbf{X}}) = \sum_{n \in \mathbb{N} \cup \{\infty\}} \mathbb{P}(T_{\mathbf{X}} \geq n)$ [8], it is not difficult to derive the central result of this section:

Theorem 20. *A PARS \mathcal{A} is (P)AST if and only if it is stochastically (P)AST.*

4 Probabilistic Term Rewrite Systems

Now we formulate probabilistic term rewriting following [5], and then lift the interpretation method for term rewriting to the probabilistic case.

We briefly recap notions from rewriting; see [4] for an introduction to rewriting. A *signature* F is a set of *function symbols* \mathbf{f} associated with their *arity* $\text{ar}(\mathbf{f}) \in \mathbb{N}$. The set $T(F, V)$ of *terms* over a signature F and a set V of variables (disjoint with F) is the least set such that $x \in T(F, V)$ if $x \in V$ and $\mathbf{f}(t_1, \dots, t_{\text{ar}(\mathbf{f})}) \in T(F, V)$ whenever $\mathbf{f} \in F$ and $t_i \in T(F, V)$ for all $1 \leq i \leq \text{ar}(\mathbf{f})$. A *substitution* is a mapping $\sigma : V \rightarrow T(F, V)$, which is extended homomorphically to terms. We write $t\sigma$ instead of $\sigma(t)$. A *context* is a term $C \in T(F, V \cup \{\square\})$ containing exactly one occurrence of a special variable \square . With $C[t]$ we denote the term obtained by replacing \square in C with t . We extend substitutions and contexts to multidistributions: $\mu\sigma := \{\{p_1 : t_1\sigma, \dots, p_n : t_n\sigma\}\}$ and $C[\mu] := \{\{p_1 : C[t_1], \dots, p_n : C[t_n]\}\}$ for $\mu = \{\{p_1 : t_1, \dots, p_n : t_n\}\}$. Given a multidistribution μ over A , we define a mapping $\bar{\mu} : A \rightarrow \mathbb{R}_{\geq 0}$ by $\bar{\mu}(a) := \sum_{p:a \in \mu} p$, which forms a distribution if $|\mu| = 1$.

Definition 21 (Probabilistic Term Rewriting). *A probabilistic rewrite rule is a pair of $l \in T(F, V)$ and $d \in \text{FDist}(T(F, V))$, written $l \rightarrow d$. A probabilistic term rewrite system (PTRS) is a (typically finite) set of probabilistic rewrite rules. We write $\widehat{\mathcal{R}}$ for the PARS consisting of a probabilistic reduction $C[l\sigma] \rightarrow C[\overline{d\sigma}]$ for every probabilistic rewrite rule $l \rightarrow d \in \mathcal{R}$, context C , and substitution σ . We say a PTRS \mathcal{R} is AST/SAST if $\widehat{\mathcal{R}}$ is.*

Note that, for a distribution d over terms, $d\sigma$ is in general a multidistribution; e.g., consider $\{\frac{1}{2}:x, \frac{1}{2}:y\}\sigma$ with $x\sigma = y\sigma$. This explains why we use $C[\overline{d\sigma}]$, which is a distribution, to obtain a probabilistic reduction above.

Example 22. The random walk of Example 2 can be modeled by a PTRS consisting of a single rule $\mathbf{s}(x) \rightarrow \{p : x, 1 - p : \mathbf{s}(\mathbf{s}(x))\}$. To rewrite a term, there are typically multiple choices of a subterm to reduce (i.e., *redexes*). For instance, $\mathbf{s}(\mathbf{f}(\mathbf{s}(0)))$ has two redexes and consequently two possible reducts:

$$\{\{p : \mathbf{f}(\mathbf{s}(0)), 1 - p : \mathbf{s}(\mathbf{f}(\mathbf{s}(0)))\}\} \quad \text{and} \quad \{\{p : \mathbf{s}(\mathbf{f}(0)), 1 - p : \mathbf{s}(\mathbf{f}(\mathbf{s}(\mathbf{s}(0))))\}\}.$$

4.1 Interpretation Methods for Proving SAST

We now generalise the *interpretation method* for term rewrite systems to the probabilistic setting. The following notion is standard.

Definition 23 (F -Algebra). *An F -algebra \mathcal{X} on a non-empty carrier set X specifies the interpretation $\mathbf{f}_{\mathcal{X}} : X^{\text{ar}(\mathbf{f})} \rightarrow X$ of each function symbol $\mathbf{f} \in F$. We say \mathcal{X} is monotone with respect to a binary relation $\succ \subseteq X \times X$ if $x \succ y$ implies $\mathbf{f}_{\mathcal{X}}(\dots, x, \dots) \succ \mathbf{f}_{\mathcal{X}}(\dots, y, \dots)$ for every $\mathbf{f} \in F$. Given an assignment $\alpha : V \rightarrow X$, the interpretation of a term is defined as follows:*

$$\llbracket t \rrbracket_{\mathcal{X}}^{\alpha} := \begin{cases} \alpha(t) & \text{if } t \in V, \\ \mathbf{f}_{\mathcal{X}}(\llbracket t_1 \rrbracket_{\mathcal{X}}^{\alpha}, \dots, \llbracket t_n \rrbracket_{\mathcal{X}}^{\alpha}) & \text{if } t = \mathbf{f}(t_1, \dots, t_n). \end{cases}$$

We write $s \succ_{\mathcal{X}} t$ iff $\llbracket s \rrbracket_{\mathcal{X}}^{\alpha} \succ \llbracket t \rrbracket_{\mathcal{X}}^{\alpha}$ for every assignment α .

Theorem 24 (cf. [27]). *A TRS \mathcal{R} is terminating iff there exists an F -algebra \mathcal{X} which is monotone with respect to a well-founded order \succ and satisfies $\mathcal{R} \subseteq \succ_{\mathcal{X}}$.*

In a proof of the completeness of the above theorem, the *term algebra* \mathcal{T} , an F -algebra on $T(F, V)$ such that $\mathbf{f}_{\mathcal{T}}(t_1, \dots, t_n) := \mathbf{f}(t_1, \dots, t_n)$, plays a crucial role. In this term algebra, assignments are substitutions, and $\llbracket t \rrbracket_{\mathcal{T}}^{\sigma} = t\sigma$. We will also use the term algebra when proving the completeness of the probabilistic version of interpretation method for proving SAST.

The following definition gives our probabilistic version of the interpretation method. It is sound and complete for proving SAST. To achieve completeness, we first keep the technique as general as possible. For an F -algebra \mathcal{X} , we lift the interpretation of terms to multidistributions as before, i.e.,

$$\llbracket \{p_1 : t_1, \dots, p_n : t_n\} \rrbracket_{\mathcal{X}}^{\alpha} := \{p_1 : \llbracket t_1 \rrbracket_{\mathcal{X}}^{\alpha}, \dots, p_n : \llbracket t_n \rrbracket_{\mathcal{X}}^{\alpha}\}.$$

Definition 25 (Probabilistic F -Algebra). *A probabilistic monotone F -algebra $(\mathcal{X}, \sqsupseteq)$ is an F -algebra \mathcal{X} equipped with a relation $\sqsupseteq \subseteq X \times \text{FDist}(X)$, such that for every $\mathbf{f} \in F$, $\mathbf{f}_{\mathcal{X}}$ is monotone with respect to \sqsupseteq , i.e., $x \sqsupseteq d$ implies $\mathbf{f}_{\mathcal{X}}(\dots, x, \dots) \sqsupseteq \mathbf{f}_{\mathcal{X}}(\dots, d, \dots)$ where $\mathbf{f}_{\mathcal{X}}(\dots, \cdot, \dots)$ is extended to (multi-) distributions. We say it is collapsible (cf. [19]) if there exist a function $G : X \rightarrow \mathbb{R}_{\geq 0}$ and $\epsilon > 0$ such that $x \sqsupseteq d$ implies $G(x) >_{\epsilon} \mathbb{E}(G(d))$.*

For a relation $\sqsupseteq \subseteq X \times \text{FDist}(X)$, we define the relation $\sqsupseteq_{\mathcal{X}} \subseteq T(F, V) \times \text{FDist}(T(F, V))$ by $t \sqsupseteq_{\mathcal{X}} d$ iff $\llbracket t \rrbracket_{\mathcal{X}}^{\alpha} \sqsupseteq \llbracket d \rrbracket_{\mathcal{X}}^{\alpha}$ for every assignment $\alpha : V \rightarrow X$. The following property is easily proven by induction.

Lemma 26. *Let $(\mathcal{X}, \sqsupseteq)$ be a probabilistic monotone F -algebra. If $s \sqsupseteq_{\mathcal{X}} d$ then $\llbracket s\sigma \rrbracket_{\mathcal{X}}^{\alpha} \sqsupseteq \llbracket d\sigma \rrbracket_{\mathcal{X}}^{\alpha}$ and $\llbracket C[s] \rrbracket_{\mathcal{X}}^{\alpha} \sqsupseteq \llbracket C[d] \rrbracket_{\mathcal{X}}^{\alpha}$ for arbitrary α, σ , and C .*

Theorem 27 (Soundness and Completeness). *A PTRS \mathcal{R} is SAST iff there exists a collapsible monotone F -algebra $(\mathcal{X}, \sqsupseteq)$ such that $\mathcal{R} \subseteq \sqsupseteq_{\mathcal{X}}$.*

Proof. For the “if” direction, we show that the PARS $\widehat{\mathcal{R}}$ is SAST using Theorem 15. Let $\alpha : V \rightarrow X$ be an arbitrary assignment, which exists as X is non-empty. Consider $s \rightarrow d \in \widehat{\mathcal{R}}$. Then we have $s = C[l\sigma]$ and $d = C[d'\sigma]$ for some σ, C , and $l \rightarrow d' \in \mathcal{R}$. By assumption we have $l \sqsupseteq_{\mathcal{X}} d'$, and thus $\llbracket s \rrbracket_{\mathcal{X}}^{\alpha} \sqsupseteq \llbracket d \rrbracket_{\mathcal{X}}^{\alpha}$ by Lemma 26. The collapsibility of \sqsupseteq gives a function $G : X \rightarrow \mathbb{R}_{\geq 0}$ and $\epsilon > 0$ such that $G(\llbracket s \rrbracket_{\mathcal{X}}^{\alpha}) >_{\epsilon} \mathbb{E}(G(\llbracket d \rrbracket_{\mathcal{X}}^{\alpha}))$, and by extending definitions we easily see $\mathbb{E}(G(\llbracket d \rrbracket_{\mathcal{X}}^{\alpha})) = \mathbb{E}(G(\llbracket d \rrbracket_{\mathcal{X}}^{\alpha}))$. Thus $G(\llbracket \cdot \rrbracket_{\mathcal{X}}^{\alpha})$ is a ranking function.

For the “only if” direction, suppose that \mathcal{R} is SAST. We show $(\mathcal{T}, \widehat{\mathcal{R}})$ forms a collapsible probabilistic monotone F -algebra orienting \mathcal{R} .

- Since \mathcal{R} is SAST, Theorem 15 gives a ranking function $f : T(F, V) \rightarrow \mathbb{R}_{\geq 0}$ and $\epsilon > 0$ for the underlying PARS $\widehat{\mathcal{R}}$. Taking $G = f$, $\widehat{\mathcal{R}}$ is collapsible.
- Suppose $s \widehat{\mathcal{R}} d$. Then we have $s = C[l\sigma]$ and $d = C[d'\sigma]$ for some C, σ , and $l \rightarrow d' \in \mathcal{R}$. As $f(\dots, C, \dots)$ is also a context, $f(\dots, s, \dots) \widehat{\mathcal{R}} f(\dots, d, \dots)$, concluding monotonicity.

- For every probabilistic rewrite rule $l \rightarrow d \in \mathcal{R}$ and every assignment (i.e., substitution) $\sigma : V \rightarrow T(F, V)$, we have $\llbracket l \rrbracket_{\mathcal{T}}^{\sigma} = l\sigma \widehat{\mathcal{R}} \overline{d\sigma} = \overline{\llbracket d \rrbracket_{\mathcal{T}}^{\sigma}}$, and hence $l \widehat{\mathcal{R}}_{\mathcal{T}} d$. This concludes $\mathcal{R} \subseteq \widehat{\mathcal{R}}_{\mathcal{T}}$. \square

4.2 Barycentric Algebras

As probabilistic F -algebras are defined so generally, it is not yet clear how to search them for ones that prove the termination of a given PTRS. Now we make one step towards finding probabilistic algebras, by imposing some conditions to (non-probabilistic) F -algebras, so that the relation \sqsubset can be defined from orderings which we are more familiar with.

Definition 28 (Barycentric Domain). A barycentric domain is a set X equipped with the barycentric operation $\mathbb{E}_X : \text{FDist}(X) \rightarrow X$.

Of particular interest in this work will be the barycentric domains $\mathbb{R}_{\geq 0}$ and $\mathbb{R}_{\geq 0}^m$ with barycentric operations $\mathbb{E}(\{p_1 : a_1, \dots, p_n : a_n\}) = \sum_{i=1}^n p_i \cdot a_i$.

We naturally generalize the following notions from standard mathematics.

Definition 29 (Concavity, Affinity). Let $f : X \rightarrow Y$ be a function from and to barycentric domains. We say f is concave with respect to an order \succ on Y if $f(\mathbb{E}_X(d)) \succcurlyeq \mathbb{E}_Y(\overline{f(d)})$ where \succcurlyeq is the reflexive closure of \succ . We say f is affine if it satisfies $f(\mathbb{E}_X(d)) = \mathbb{E}_Y(\overline{f(d)})$.

Clearly, every affine function is concave.

Now we arrive at the main definition and theorem of this section.

Definition 30 (Barycentric F -Algebra). A barycentric F -algebra is a pair (\mathcal{X}, \succ) of an F -algebra \mathcal{X} on a barycentric domain X and an order \succ on X , such that for every $\mathbf{f} \in F$, $\mathbf{f}_{\mathcal{X}}$ is monotone and concave with respect to \succ . We say it is collapsible if there exist a concave function $\mathbf{G} : X \rightarrow \mathbb{R}_{\geq 0}$ (with respect to \succ) and $\epsilon > 0$ such that $\mathbf{G}(x) >_{\epsilon} \mathbf{G}(y)$ whenever $x \succ y$.

We define the relation $\succ^{\mathbb{E}} \subseteq X \times \text{FDist}(X)$ by $x \succ^{\mathbb{E}} d$ iff $x \succ \mathbb{E}_X(d)$.

Note that the following theorem claims soundness but not completeness, in contrast to Theorem 27.

Theorem 31. A PTRS \mathcal{R} is SAST if $\mathcal{R} \subseteq \succ^{\mathbb{E}}_{\mathcal{X}}$ for a collapsible barycentric F -algebra (\mathcal{X}, \succ) .

Proof. Due to Theorem 27, it suffices to show that $(\mathcal{X}, \succ^{\mathbb{E}})$ is a collapsible probabilistic monotone F -algebra. Concerning monotonicity, suppose $x \succ^{\mathbb{E}} d$, i.e., $x \succ \mathbb{E}_X(d)$, and let $\mathbf{f} \in F$. Since $\mathbf{f}_{\mathcal{X}}$ is monotone and concave with respect to \succ in every argument, we have

$$\mathbf{f}_{\mathcal{X}}(\dots, x, \dots) \succ \mathbf{f}_{\mathcal{X}}(\dots, \mathbb{E}_X(d), \dots) \succcurlyeq \mathbb{E}_X(\overline{\mathbf{f}_{\mathcal{X}}(\dots, d, \dots)}) .$$

Concerning collapsibility, whenever $x \succ \mathbb{E}_{\mathcal{X}}(d)$ we have

$$\begin{aligned} \mathbf{G}(x) &>_{\epsilon} \mathbf{G}(\mathbb{E}_{\mathcal{X}}(d)) && \text{by assumption on } \mathbf{G}, \\ &\geq \mathbb{E}(\overline{\mathbf{G}(d)}) && \text{as } \mathbf{G} : X \rightarrow \mathbb{R} \text{ is concave with respect to } >, \\ &= \mathbb{E}(\mathbf{G}(d)) && \text{by the definition of } \mathbb{E} \text{ on multidistributions.} \quad \square \end{aligned}$$

The rest of the section recasts two popular interpretation methods, polynomial and matrix interpretations (over the reals), as barycentric F -algebras.

Polynomial interpretations were introduced (on natural numbers [21] and real numbers [22]) for the termination analysis of non-probabilistic rewrite systems. Various techniques for synthesizing polynomial interpretations (e.g., [15]) exist, and these techniques are easily applicable in our setting.

Definition 32 (Polynomial Interpretation). A polynomial interpretation is an F -algebra \mathcal{X} on $\mathbb{R}_{\geq 0}$ such that $\mathbf{f}_{\mathcal{X}}$ is a polynomial for every $\mathbf{f} \in F$. We say \mathcal{X} is multilinear if every $\mathbf{f}_{\mathcal{X}}$ is of the following form with $\mathbf{c}_V \in \mathbb{R}_{\geq 0}$:

$$\mathbf{f}_{\mathcal{X}}(x_1, \dots, x_n) = \sum_{V \subseteq \{x_1, \dots, x_n\}} \mathbf{c}_V \cdot \prod_{x_i \in V} x_i.$$

In order to use polynomial interpretations for probabilistic termination, multilinearity is necessary for satisfying the concavity condition.

Proposition 33. Let \mathcal{X} be a monotone multilinear polynomial interpretation and $\epsilon > 0$. If $\llbracket l \rrbracket_{\mathcal{X}}^{\alpha} >_{\epsilon} \mathbb{E}(\llbracket d \rrbracket_{\mathcal{X}}^{\alpha})$ for every $l \rightarrow d \in \mathcal{R}$ and α , then the PTRS \mathcal{R} is SAST.

Proof. The order $>_{\epsilon}$ is trivially collapsible with $\mathbf{G}(x) = x$. Further, every multilinear polynomial is affine and thus concave in all variables. Hence $(\mathcal{X}, >_{\epsilon})$ forms a barycentric F -algebra, and thus Theorem 31 shows that \mathcal{R} is SAST. \square

An observation by Lucas [22] also holds in probabilistic case: To prove a finite PTRS \mathcal{R} SAST with polynomial interpretations, we do not have to find ϵ , but it is sufficient to check $l >_{\mathbb{E}_{\mathcal{X}}}^{\alpha} d$ for all rules $l \rightarrow d \in \mathcal{R}$. Define $\epsilon_{l \rightarrow d} := \mathbb{E}(\llbracket d \rrbracket_{\mathcal{X}}^{\alpha}) - \llbracket l \rrbracket_{\mathcal{X}}^{\alpha}$ for such α that $\alpha(x) = 0$. Then for any other α , we can show $\mathbb{E}(\llbracket d \rrbracket_{\mathcal{X}}^{\alpha}) - \llbracket l \rrbracket_{\mathcal{X}}^{\alpha} \geq \epsilon_{l \rightarrow d} > 0$. As \mathcal{R} is finite, we can take $\epsilon := \min\{\epsilon_{l \rightarrow d} \mid l \rightarrow d \in \mathcal{R}\} > 0$.

Example 34 (Example 22 Continued). Consider again the PTRS consisting of the single rule $\mathbf{s}(x) \rightarrow \{p : x, 1 - p : \mathbf{s}(\mathbf{s}(x))\}$. Define the polynomial interpretation \mathcal{X} by $0_{\mathcal{X}} := 0$ and $\mathbf{s}_{\mathcal{X}}(x) := x + 1$. Then whenever $p > \frac{1}{2}$ we have

$$\llbracket \mathbf{s}(x) \rrbracket_{\mathcal{X}}^{\alpha} = x + 1 > p \cdot x + (1 - p) \cdot (x + 2) = \mathbb{E}(\llbracket \{p : x, 1 - p : \mathbf{s}(\mathbf{s}(x))\} \rrbracket_{\mathcal{X}}^{\alpha}).$$

Thus, when $p > \frac{1}{2}$ the PTRS is SAST by Proposition 33.

We remark that polynomial interpretations are not covered by [5, Theorem 5], since *context decrease* [5, Definition 8] demands $\llbracket \mathbf{f}(t) \rrbracket_{\mathcal{X}}^{\alpha} - \llbracket \mathbf{f}(t') \rrbracket_{\mathcal{X}}^{\alpha} \leq \llbracket t \rrbracket_{\mathcal{X}}^{\alpha} - \llbracket t' \rrbracket_{\mathcal{X}}^{\alpha}$, which excludes interpretations such as $\mathbf{f}_{\mathcal{X}}(x) = 2x$.

Matrix interpretations are introduced for the termination analysis of term rewriting [13]. Now we extend them for probabilistic term rewriting.

Definition 35 (Matrix Interpretation). A (real) matrix interpretation is an F -algebra \mathcal{X} on $\mathbb{R}_{\geq 0}^m$ such that for every $f \in F$, $\mathbf{f}_{\mathcal{X}}$ is of the form

$$\mathbf{f}_{\mathcal{X}}(\mathbf{x}_1, \dots, \mathbf{x}_n) = \sum_{i=1}^n C_i \cdot \mathbf{x}_i + \mathbf{c}, \quad (1)$$

where $\mathbf{c} \in \mathbb{R}_{\geq 0}^m$, and $C_i \in \mathbb{R}_{\geq 0}^{m \times m}$. The order $\gg_{\epsilon} \subseteq \mathbb{R}_{\geq 0}^m \times \mathbb{R}_{\geq 0}^m$ is defined by

$$(x_1, \dots, x_m)^T \gg_{\epsilon} (y_1, \dots, y_m)^T :\iff x_1 >_{\epsilon} y_1 \text{ and } x_i \geq y_i \text{ for all } i = 2, \dots, m.$$

It is easy to derive the following from Theorem 31:

Proposition 36. Let \mathcal{X} be a monotone matrix interpretation and $\epsilon > 0$. If $\llbracket l \rrbracket_{\mathcal{X}}^{\alpha} \gg_{\epsilon} \mathbb{E}(\llbracket d \rrbracket_{\mathcal{X}}^{\alpha})$ for every $l \rightarrow d \in \mathcal{R}$ and α , then the PTRS \mathcal{R} is SAST.

As in polynomial interpretations, for finite systems we do not have to find ϵ . Monotonicity can be ensured if (1) satisfies $(C_i)_{1,1} \geq 1$ for all i , cf. [13].

Example 37. Consider the PTRS consisting of the single probabilistic rule

$$\mathbf{a}(\mathbf{a}(x)) \rightarrow \{p : \mathbf{a}(\mathbf{a}(\mathbf{a}(x))), 1 - p : \mathbf{a}(\mathbf{b}(\mathbf{a}(x)))\}.$$

Consider the two-dimensional matrix interpretation

$$\llbracket \mathbf{a} \rrbracket(\mathbf{x}) = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \cdot \mathbf{x} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \llbracket \mathbf{b} \rrbracket(\mathbf{x}) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \cdot \mathbf{x}.$$

Then we have

$$\begin{aligned} \llbracket \mathbf{a}(\mathbf{a}(x)) \rrbracket^{\alpha} &= \begin{bmatrix} x_1 + x_2 + 1 \\ 1 \end{bmatrix} \gg_{1-2p} \begin{bmatrix} x_1 + x_2 + 2p \\ 1 \end{bmatrix} \\ &= p \cdot \llbracket \mathbf{a}(\mathbf{a}(\mathbf{a}(x))) \rrbracket^{\alpha} + (1 - p) \cdot \llbracket \mathbf{a}(\mathbf{b}(\mathbf{a}(x))) \rrbracket^{\alpha} \end{aligned}$$

where $\alpha(x) = (x_1, x_2)^T$. Hence this PARS is SAST if $p < \frac{1}{2}$, by Proposition 36.

It is worthy of note that the above example cannot be handled with polynomial interpretations, intuitively because monotonicity enforces the interpretation of the probable reducts $\mathbf{a}(\mathbf{a}(\mathbf{a}(x)))$ and $\mathbf{a}(\mathbf{b}(\mathbf{a}(x)))$ to be greater than that of the left-hand side $\mathbf{a}(\mathbf{a}(x))$. Generally, polynomial and matrix interpretations are incomparable in strength.

5 Conclusion

This is a study on how much of the classic interpretation-based techniques well known in term rewriting can be extended to *probabilistic* term rewriting, and to what extent they remain automatable. The obtained results are quite encouraging, although finding ways to combine techniques is crucial if one wants to capture a reasonably large class of systems, similarly to what happens in ordinary term rewriting [2]. Another hopeful future work includes extending our result for proving AST, not only SAST.

We extended the termination prover NaTT [28] with a syntax for probabilistic rules, and implemented the probabilistic versions of polynomial and matrix interpretations. For usage and implementation details, we refer to the extended version of this paper. Here we only report that we tested the implementation on the examples presented in the paper and successfully found termination proofs.

The following example would deserve some attention.

Example 38. Consider the following encoding of [14, Fig. 1]:

$$\begin{array}{ll} ?(x) \rightarrow \{\frac{1}{2} : ?(\mathbf{s}(x)), \frac{1}{2} : \$(\mathbf{g}(x))\} & \$(0) \rightarrow \{1 : 0\} \\ ?(x) \rightarrow \{1 : \$(\mathbf{f}(x))\} & \$(\mathbf{s}(x)) \rightarrow \{1 : \$(x)\} \end{array}$$

describing a game where the player (strategy) can choose either to quit the game and ensure prize $\$(\mathbf{f}(x))$, or to try a coin-toss which on success increments the score and on failure ends the game with consolation prize $\$(\mathbf{g}(x))$.

When \mathbf{f} and \mathbf{g} can be bounded by linear polynomials, it is possible to automatically prove that the system is SAST. For instance, with rules for $\mathbf{f}(x) = 2x$ and $\mathbf{g}(x) = \lfloor \frac{x}{2} \rfloor$, NaTT (combined with the SMT solver z3 version 4.4.1) found the following polynomial interpretation proving SAST:

$$\begin{array}{lll} ?_{\mathcal{X}}(x) = 7x + 11 & \mathbf{s}_{\mathcal{X}}(x) = x + 1 & 0_{\mathcal{X}} = 1 \\ \mathbf{f}_{\mathcal{X}}(x) = 3x + 1 & \mathbf{g}_{\mathcal{X}}(x) = 2x + 1 & \$_{\mathcal{X}}(x) = 2x + 1 . \end{array}$$

Acknowledgments. We thank the anonymous reviewers for their constructive remarks that improved the paper. Example 12 is due to one of them. We thank Luis María Ferrer Fioriti for the analysis of a counterexample in [14]. This work is partially supported by the ANR projects 14CE250005 ELICA and 16CE250011 REPAS, the FWF project Y757, and JST ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603).

References

1. Agha, G., Meseguer, J., Sen, K.: PMAude: rewrite-based specification language for probabilistic object systems. *Electr. Notes Theor. Comput. Sci.* **153**(2), 213–239 (2006)
2. Avanzini, M.: Verifying polytime computability automatically. Ph.D. thesis, University of Innsbruck (2013)

3. Avanzini, M., Dal Lago, U., Yamada, A.: On probabilistic term rewriting (Technical report). CoRR cs/CC/1802.09774 (2018). <http://www.arxiv.org/abs/1802.09774>
4. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
5. Bournez, O., Garnier, F.: Proving positive almost-sure termination. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 323–337. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_24
6. Bournez, O., Garnier, F.: Proving positive almost sure termination under strategies. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 357–371. Springer, Heidelberg (2006). https://doi.org/10.1007/11805618_27
7. Bournez, O., Kirchner, C.: Probabilistic rewrite strategies. Applications to ELAN. In: Proceedings of 13th RTA, pp. 252–266 (2002)
8. Brémaud, P.: *Marcov Chains*. Springer, New York (1999). <https://doi.org/10.1007/978-1-4757-3124-8>
9. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through Positivstellensatz’s. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 3–22. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_1
10. Dal Lago, U., Zorzi, M.: Probabilistic operational semantics for the lambda calculus. *RAIRO - TIA* **46**(3), 413–450 (2012)
11. Dal Lago, U., Grellois, C.: Probabilistic termination by monadic affine sized typing. In: Proceedings of 26th ESOP, pp. 393–419 (2017)
12. Dal Lago, U., Martini, S.: On constructor rewrite systems and the lambda calculus. *LMCS* **8**(3), 1–27 (2012)
13. Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of term rewriting. *JAR* **40**(3), 195–220 (2008)
14. Ferrer Fioriti, L.M., Hermanns, H.: Probabilistic termination: soundness, completeness, and compositionality. In: Proceedings of 42nd POPL, pp. 489–501. ACM (2015)
15. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT solving for termination analysis with polynomial interpretations. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 340–354. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72788-0_33
16. Gnaedig, I.: Induction for positive almost sure termination. In: PPDP 2017, pp. 167–178. ACM (2007)
17. Goldwasser, S., Micali, S.: Probabilistic encryption. *JCSS* **28**(2), 270–299 (1984)
18. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. In: Proceedings of 24th UAI, pp. 220–229. AUAI Press (2008)
19. Hirokawa, N., Moser, G.: Automated complexity analysis based on context-sensitive rewriting. In: Dowek, G. (ed.) RTA 2014. LNCS, vol. 8560, pp. 257–271. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_18
20. Kaminski, B.L., Katoen, J.: On the hardness of almost-sure termination. In: MFCS 2015, Proceedings, Part I, Milan, Italy, 24–28 August 2015, pp. 307–318 (2015)
21. Lankford, D.: Canonical algebraic simplification in computational logic. Technical report ATP-25, University of Texas (1975)
22. Lucas, S.: Polynomials over the reals in proofs of termination: from theory to practice. *ITA* **39**(3), 547–586 (2005)
23. Puterman, M.L.: *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, 1st edn. Wiley, New York (1994)

24. Rabin, M.O.: Probabilistic automata. *Inf. Control* **6**(3), 230–245 (1963)
25. Saheb-Djahromi, N.: Probabilistic LCF. In: MFCS, pp. 442–451 (1978)
26. Santos, E.S.: Probabilistic turing machines and computability. *Proc. Am. Math. Soc.* **22**(3), 704–710 (1969)
27. Terese (ed.): *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
28. Yamada, A., Kusakari, K., Sakabe, T.: Nagoya termination tool. In: Dowek, G. (ed.) RTA 2014. LNCS, vol. 8560, pp. 466–475. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_32



Equivalence Checking of Non-deterministic Operations

Sergio Antoy¹  and Michael Hanus²  

¹ Computer Science Department, Portland State University, Portland, OR, USA
antoy@cs.pdx.edu

² Institut für Informatik, CAU Kiel, 24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract. Checking the semantic equivalence of operations is an important task in software development. For instance, regression testing is a routine task when software systems are developed and improved, and software package managers require the equivalence of operations in different versions of a package within the same major version. In order to support a good automation of this process, a solid foundation is required. It has been shown that the notion of equivalence is not obvious when non-deterministic features are present. In this paper, we discuss a general notion of equivalence in functional logic programs and develop a practical method to check it. Our method can be integrated in a property-based testing tool which is used in a software package manager to check the semantic versioning of software packages.

1 Motivation

Functional logic languages combine the most important features of functional and logic programming in a single language (see [4, 14] for recent surveys). In this paper we consider Curry [18], a contemporary functional logic language which conceptually extends Haskell with common features of logic programming. Hence, Curry combines the demand-driven evaluation of functions with non-deterministic evaluation of operations defined by overlapping rules. As discussed in [6], the combination of these features poses new issues for defining the equivalence of expressions. Actually, three different notions of equivalence can be distinguished:

1. *Ground equivalence*: Two expressions are equivalent if they have the same results when their variables are replaced by ground terms.
2. *Computed-result equivalence*: Two expressions are equivalent if they have the same outcomes, i.e., variables in expressions are considered as free variables which might be instantiated during the evaluation process.
3. *Contextual equivalence*: Two expressions are equivalent if they produce the same outcomes in all possible contexts.

Ground equivalence seems reasonable for functional programs since free variables are not allowed in expressions to be evaluated in functional programming. For instance, consider the Boolean negation defined by

```
not False = True
not True  = False
```

The expressions `not (not x)` and `x` are ground equivalent, which can be checked easily by instantiating `x` to `True` and `False`, respectively, and evaluating both expressions. However, these expressions are not computed-result equivalent w.r.t. the narrowing semantics of functional logic programming: the expression `not (not x)` evaluates to the two outcomes $\{x = \text{False}\} \text{False}$ and $\{x = \text{True}\} \text{True}$,¹ whereas the expression `x` evaluates to the single result $\{\} x$ without instantiating the free variable `x`. Due to these differences, Bacci et al. [6] states that ground equivalence is “the (only possible) equivalence notion used in the pure functional paradigm.” As we will see later, this is not true since contextual equivalence is also relevant in non-strict functional languages.

The previous example shows that the evaluation of ground equivalent expressions might result in answers with different degrees of instantiation. However, the presence of logic variables and non-determinism might also lead to different results when ground equivalent expressions are put in a same context. For instance, consider the following contrived example [6] (a more natural example will be shown later):

```
f x = C (h x)           g A = C A
h A = A
```

The expressions `f x` and `g x` are computed-result equivalent since the only computed result is $\{x = A\} C A$. Now consider the following operation:

```
k (C x) B = B
```

Then the expression `k (f x) x` evaluated lazily produces $\{x = B\} B$, whereas the expression `k (g x) x` produces no values. In fact, the evaluation of `g x` instantiates (narrows) `x` to `A`, and `k (C A) A` is irreducible. Hence, the ground and computed-result equivalent expressions are not contextually equivalent.

The equivalence of operations is important when existing software packages are further developed, e.g., refactored or implemented with more efficient data structures. In this case, we want to ensure that operations available in the API of both versions of a software package are equivalent, as long as we do not introduce intended API changes. For this purpose, software package management systems associate version numbers to software packages. In the semantic versioning standard,² a version number consists of major, minor, and patch number, separated by dots, and an optional pre-release specifier. For instance, `2.0.1` and `3.2.1-alpha.2` are valid version numbers. An intended and incompatible change of API operations is marked by a change in the major version number. Thus, operations available in two versions of a package with identical major

¹ Note that functional logic languages compute a substitution as well as a value as a result.

² <http://www.semver.org>.

version numbers should be equivalent. Unfortunately, most package managers do not check this equivalence but leave it as a recommendation to the package developer.

Improving this situation is the motivation for our work. We want to develop a tool to check the equivalence of two operations. Since we aim to integrate this kind of semantic versioning checking in a practical software package manager [16], the tool should be fully automatic. Thus, we are going to test equivalence properties rather than verify them. Although this might be unsatisfactory from a theoretical point of view, it could be quite powerful from a practical point of view and might prevent wasting time to prove incorrect properties. For instance, property-based test tools like QuickCheck [8] provide great confidence in programs by checking program properties with many test inputs. For instance, we could check the equivalence of two operations f and f' by checking the equation $f x = f' x$ with many values for x . The previous discussion of equivalence criteria shows that this property checks only the ground equivalence of f and f' . However, in the context of semantic versioning checking, ground equivalence is too restricted since equivalent operations should deliver the same results in any context. Therefore, contextual equivalence is desired. Actually, this kind of equivalence has been proposed in [5] as the only notion to state the correctness of an implementation w.r.t. a specification in functional logic programming. Unfortunately, the automatic checking of contextual equivalence with property-based test tools does not seem feasible due to the unlimited number of possible contexts. Therefore, Bacci et al. [6] state: “In a test-based approach. . . the addition of a further outer context would dramatically alter the performance.” Therefore, the authors abandon the use of a standard property-based test tool in their work.

In this paper we show that we can use such tools for contextual equivalence (and, thus, semantic versioning) checking if we use an appropriate encoding of test data. For this purpose, we develop some theoretical results that allow us to reduce the contexts to be considered for equivalence checking. From these results, we show how property-based testing can be used for this purpose. Based on these results, we extend an existing property-based test tool for functional logic programs [15] to test the equivalence of operations. This is the basis of a software package manager with semantic versioning checking [16].

In the next section, we review the main concepts of functional logic programming and Curry. Section 3 defines our notion of equivalence which is used in Sect. 4 to develop practically useful characterizations of equivalent operations. Section 5 shows how to use these criteria in a property-based testing tool. Section 6 discusses some related work before we conclude.

2 Functional Logic Programming and Curry

We briefly review those elements of functional logic languages and Curry that are necessary to understand the contents of this paper. More details can be found in surveys on functional logic programming [4, 14] and in the language report [18].

Curry is a declarative multi-paradigm language combining in a seamless way features from functional and logic programming. The syntax of Curry is close to

Haskell [23]. In addition to Haskell, Curry allows *free (logic) variables* in conditions and right-hand sides of rules. Thus, *expressions* in Curry programs contain *operations* (defined functions), *constructors* (introduced in data type declarations), and *variables* (arguments of operations or free variables). Function calls with free variables are evaluated by a possibly non-deterministic instantiation of demanded arguments [2]. In contrast to Haskell, rules with overlapping left-hand sides are non-deterministically (and not sequentially) applied.

Example 1. The following example shows the definition of a non-deterministic list insertion operation in Curry:

```
insert :: a → [a] → [a]
insert x ys      = x : ys
insert x (y:ys) = y : insert x ys
```

For instance, the expression `insert 0 [1,2]` non-deterministically evaluates to one of the values `[0,1,2]`, `[1,0,2]`, or `[1,2,0]`. Based on this operation, we can easily define permutations:

```
perm []      = []
perm (x:xs) = insert x (perm xs)
```

Thus, `perm [1,2,3,4]` non-deterministically evaluates to all 24 permutations of the input list.

Non-deterministic operations, which are interpreted as mappings from values into sets of values [13], are an important feature of contemporary functional logic languages. Using non-deterministic operations as arguments could cause a semantical ambiguity. Consider the operations

```
coin = 0
coin = 1
double x = x + x
```

Standard term rewriting produces, among others, the derivation

```
double coin → coin + coin → 0 + coin → 0 + 1 → 1
```

whose result is (presumably) unintended. Therefore, González-Moreno et al. [13] proposed the rewriting logic CRWL as a logical foundation for declarative programming with non-strict and non-deterministic operations. This logic specifies the *call-time choice* semantics [19] where values of the arguments of an operation are set, though not computed, before the operation is evaluated. In a lazy strategy, this is naturally obtained by sharing. For instance, the two occurrences of `coin` in the derivation above are shared so that “`double coin`” has only two results: 0 or 2. Since standard term rewriting does not conform to the intended call-time choice semantics, other notions of rewriting have been proposed to formalize this idea, like graph rewriting [11,12] or let rewriting [21]. In this paper, we use a simple reduction relation that we sketch without giving all details (which can be found in [21]).

In the following, we ignore free (logic) variables since they can be considered as syntactic sugar for non-deterministic data generator operations [3]. Thus, a *value* is an expression without operations or free variables. To cover

non-strict computations, expressions can also contain the special symbol \perp to represent *undefined or unevaluated values*. A *partial value* is a value containing occurrences of \perp . A *partial constructor substitution* is a substitution that replaces variables by partial values. A *context* $\mathcal{C}[\cdot]$ is an expression with some “hole”. The reduction relation we use throughout this paper is defined as follows (conditional rules are not considered for the sake of simplicity):

$$\begin{array}{ll} \text{Fun} & \mathcal{C}[f \sigma(t_1) \dots \sigma(t_n)] \rightarrow \mathcal{C}[\sigma(r)] \quad \text{where } f \ t_1 \dots t_n = r \text{ is a program rule} \\ & \quad \text{and } \sigma \text{ a partial constructor substitution} \\ \text{Bot} & \mathcal{C}[e] \rightarrow \mathcal{C}[\perp] \quad \text{where } e \neq \perp \end{array}$$

The first rule models call-time choice: if a rule is applied, the actual arguments of the operation must have been evaluated to partial values. The second rule models non-strictness by allowing the evaluation of any subexpression to an undefined value (which is intended if the value of this subexpression is not demanded). As usual, $\overset{*}{\rightarrow}$ denotes the reflexive and transitive closure of this reduction relation. The equivalence of this rewrite relation and CRWL is shown in [21].

3 Equivalent Operations

As discussed above, equivalence of operations can be defined in different ways. Ground equivalence and computed result equivalence only compare the values of applications. This is too weak since some operations have no finite values.

Example 2. Consider the following operations that generate infinite lists of numbers:

```
ints1 n = n : ints1 (n+1)      ints2 n = n : ints2 (n+2)
```

Since these operations do not produce finite values, we cannot detect any difference when comparing only computed results. However, they behave different when put into some context, e.g., an operation that selects the second element of a list:

```
snd (x:y:zs) = y
```

Now, `snd (ints1 0)` and `snd (ints2 0)` evaluate to 1 and 2, respectively.

Therefore, we do not consider these operations as equivalent. This motivates the following notion of equivalence for possibly non-terminating and non-deterministic operations.³

Definition 1 (Equivalence). *Let f_1, f_2 be operations of type $\tau \rightarrow \tau'$. f_1 is equivalent to f_2 iff, for any expression E_1 , $E_1 \overset{*}{\rightarrow} v$ iff $E_2 \overset{*}{\rightarrow} v$, where v is a value and E_2 is obtained from E_1 by replacing each occurrence of f_1 with f_2 .*

This notion of equivalence conforms with the usual notion of contextual equivalence in programming languages (e.g., see [25] for a tutorial). It was already

³ The extension to operations with several arguments is straightforward. For the sake of simplicity, we formally define our notions only for unary operations.

proposed in [5] as the notion of equivalence for functional logic programs and also defined in [6] as “contextual equivalence” for functional logic programs.

Thus, `ints1` and `ints2` are not equivalent. Moreover, even terminating operations that always compute same results might not be equivalent if put into some context.

Example 3. Consider the definition of lists sorted in ascending order:

```
sorted []           = True
sorted [_]         = True
sorted (x:y:zs)    = x<=y && sorted (y:zs)
```

We can use this definition and the definition of permutations above to provide a precise specification of sorting a list by computing some sorted permutation:

```
sort xs | sorted ys = ys    where ys = perm xs
```

We might try to obtain an even more compact formulation by defining the “sorted” property as an operation that is the (partial) identity on sorted lists:

```
idSorted []           = []
idSorted [x]         = [x]
idSorted (x:y:zs) | x<=y = x : idSorted (y:zs)
```

Then we can define another operation to sort a list by composing `perm` and `idSorted`:

```
sort' xs = idSorted (perm xs)
```

Although both `sort` and `sort'` compute sorted lists, they might behave differently in a same context. For instance, suppose we want to compute the minimum of a list by returning the head element of the sorted list:

```
head (x:xs) = x
```

Then `head (sort [3,2,1])` returns 1, as expected, but `head (sort' [3,2,1])` returns 1 as well as 2. The latter unintended value is obtained by computing the permutation `[2,3,1]` so that `head (idSorted [2,3,1])` returns 2, since the list rest `idSorted [3,1]` is not evaluated due to non-strictness.

This example shows that our strong notion of equivalence is reasonable. However, testing this equivalence might require the generation of arbitrary contexts. Therefore, we show in the next section how to avoid this context generation.

4 Refined Equivalence Criteria

The definition of equivalence as stated in Definition 1 covers the intuition that equivalent operations can be interchanged at any place in an expression without changing its value. Proving such a general form of equivalence could be difficult. Therefore, we define another form of equivalence that is based on an operation to observe the computed results of the corresponding operations.

Definition 2 (Observable equivalence). *Let f_1, f_2 be operations of type $\tau \rightarrow \tau'$. f_1 is observably equivalent to f_2 iff, for all operations g of type $\tau' \rightarrow \tau''$, all expressions e and values v , $g (f_1 e) \xrightarrow{*} v$ iff $g (f_2 e) \xrightarrow{*} v$.*

We can expect that proving observable equivalence is easier than equivalence since we trade a context made of an arbitrary expression with multiple occurrences of a function f with a single function call with a single occurrence of f . Fortunately, the next theorem shows that proving observable equivalence is sufficient in general.

Theorem 1. *Let f_1, f_2 be operations of type $\tau \rightarrow \tau'$. f_1 and f_2 are equivalent iff they are observably equivalent.*

Proof. It is trivial that equivalence implies observable equivalence. Hence, we assume that f_1 and f_2 are observably equivalent, i.e., for all operations g of type $\tau' \rightarrow \tau''$, all expressions e and values v , $g(f_1 e) \xrightarrow{*} v$ iff $g(f_2 e) \xrightarrow{*} v$. We show by induction on the number n of occurrences of the symbol f_1 the following claim:

If E_1 is an expression with n occurrences of f_1 , E_2 is obtained from E_1 by replacing each occurrence of f_1 with f_2 , and v is a value, then $E_1 \xrightarrow{*} v$ iff $E_2 \xrightarrow{*} v$.

Base case ($n = 0$): Since E_1 contains no occurrence of f_1 , $E_2 = E_1$ and the claim is trivially satisfied.

Inductive case ($n > 0$): Assume the claim holds for $n - 1$ and E_1 contains n occurrences of f_1 and $E_1 \xrightarrow{*} v$ for some value v . We have to show that $E_2 \xrightarrow{*} v$ (the opposite direction is symmetric) where E_2 is obtained from E_1 by replacing each occurrence of f_1 with f_2 . Let p be a position in E_1 with $E_1|_p = f_1 e$ and e does not contain any occurrence of f_1 . Since $E_1 \xrightarrow{*} v$, by definition of $\xrightarrow{*}$, there is a partial value t_1 with $f_1 e \xrightarrow{*} t_1$ and $E_1[t_1]_p \xrightarrow{*} v$. We define a new operation g by

$$g x = E_1[x]_p$$

where x is a variable that does not occur in E_1 . Hence $g(f_1 e) \xrightarrow{*} g t_1 \rightarrow E_1[t_1]_p \xrightarrow{*} v$. Our assumption implies $g(f_2 e) \xrightarrow{*} v$. By definition of $\xrightarrow{*}$, there is a partial value t_2 with $g(f_2 e) \xrightarrow{*} g t_2 \rightarrow E_1[t_2]_p \xrightarrow{*} v$. Since $E_1[t_2]_p$ contains $n - 1$ occurrences of f_1 , the induction hypothesis implies that $E_2[t_2]_p \xrightarrow{*} v$. Therefore, $E_2 = E_2[f_2 e]_p \xrightarrow{*} E_2[t_2]_p \xrightarrow{*} v$. \square

A proof that two operations are observably equivalent could still be difficult since we have to take all possible observation operations into account. However, the next result shows that it is sufficient to verify that two operations yield always the same partial values on identical inputs.

Theorem 2. *Let f_1, f_2 be operations of type $\tau \rightarrow \tau'$. If, for all expressions e and partial values t , $f_1 e \xrightarrow{*} t$ iff $f_2 e \xrightarrow{*} t$, then f_1 and f_2 are equivalent.*

Proof. By Theorem 1 it is sufficient to show the observable equivalence of f_1 and f_2 . Hence, let g be an operation of type $\tau' \rightarrow \tau''$, e an expression and v a value with $g(f_1 e) \xrightarrow{*} v$. We have to show that $g(f_2 e) \xrightarrow{*} v$ (the other direction is symmetric). By definition of $\xrightarrow{*}$, there is some partial value t with $f_1 e \xrightarrow{*} t$ and $g t \xrightarrow{*} v$. By the assumption of the theorem, $f_2 e \xrightarrow{*} t$. Hence, $g(f_2 e) \xrightarrow{*} g t \xrightarrow{*} v$. \square

Note that the consideration of all *partial* result values is essential to establish equivalence. For instance, consider the operations `sort` and `sort'` defined in Sect. 3. Although `sort` and `sort'` compute the same *values*, we have that `sort'` $[2,3,1] \xrightarrow{*} 2 : \perp$ but `sort` $[2,3,1]$ cannot be derived to $2 : \perp$. Actually, we have seen that `sort` and `sort'` are not equivalent.

The following result is the converse of Theorem 2. It shows that not only having the same partial values is a sufficient condition for the equivalence of function, but also a necessary condition. For partial values t and u , we write $t < u$ iff t is obtained by one or more applications of the `Bot` rule to u . It follows that if u is a partial value of an expression e , then any $t < u$ is also a partial value of e . If t is a partial value, we denote by \bar{t} an expression obtained from t by replacing any instance of \perp in t with a fresh variable.

Theorem 3. *Let f_1, f_2 be operations of type $\tau \rightarrow \tau'$. If, for some expression e , the partial values of $f_1 e$ differ from those of $f_2 e$, then f_1 and f_2 are not equivalent.*

Proof. We construct a function g that, under the statement hypothesis, witnesses the non-equivalence of f_1 and f_2 . Let T_1 be the set of partial values of $f_1 e$ and T_2 the set of partial values of $f_2 e$. W.l.o.g., we assume that there exists some partial value $t \in T_1$ such that $t \notin T_2$. Let g be defined by the single rule:

$$g \bar{t} \rightarrow 0$$

Then, $g (f_1 e) \xrightarrow{*} g t \rightarrow 0$, whereas we show that $g (f_2 e) \not\xrightarrow{*} 0$. Suppose the contrary. Then, it must be that $f_2 e \xrightarrow{*} u$ with u is an instance of \bar{t} . This implies $t < u$, which in turn implies $t \in T_2$. \square

The next corollary is useful to avoid the consideration of all argument expressions in equivalence proofs.

Corollary 1. *Let f_1, f_2 be operations of type $\tau \rightarrow \tau'$. If, for all partial values t and t' , $f_1 t \xrightarrow{*} t'$ iff $f_2 t \xrightarrow{*} t'$, then f_1 and f_2 are equivalent.*

Proof. Assume that $f_1 t \xrightarrow{*} t'$ iff $f_2 t \xrightarrow{*} t'$ holds for all partial values t and t' . Consider an expression e and a partial value t_1 such that $f_1 e \xrightarrow{*} t_1$. By definition of $\xrightarrow{*}$, there is a partial value t_0 with $e \xrightarrow{*} t_0$ and $f_1 t_0 \xrightarrow{*} t_1$. Our assumption implies $f_2 t_0 \xrightarrow{*} t_1$. Hence $f_2 e \xrightarrow{*} f_2 t_0 \xrightarrow{*} t_1$. Since the other direction is symmetric, Theorem 2 implies the equivalence of f_1 and f_2 . \square

Hence, we have a sufficient criterion for equivalence checking which does not require the enumeration of arbitrary contexts. Instead, it is sufficient to test the equivalence on all partial values. Such a test can be performed by property-based test tools, as shown in the next section.

One may wonder whether the consideration of values instead of partial values is enough for equivalence checking. The next example shows that the answer is negative.

Example 4. Consider the following operations that take and return Booleans.

```

f1 True  = True
f1 False = True
f2 _    = True

```

Functions `f1` and `f2` behave identically on every input *value*. However, `f1 ⊥` has no value, whereas `f2 ⊥` has value `True`. Thus, values as arguments are not as discriminating as partial values to expose a difference in behavior, whereas partial values are as discriminating as expressions. Actually, `f1` and `f2` are not equivalent: consider the operation `failed` which has no value.⁴ Then `f2 failed` has value `True` whereas `f1 failed` has no value.

Corollary 1 requires to compare all partial result values and not just computed results. The former is more laborious since an expression might evaluate to many partial values even if it has a single value. For instance, consider the list generator

```

fromTo m n = if m>n then [] else m : fromTo (m+1) n

```

The expression `fromTo 1 5` evaluates to the single value `[1,2,3,4,5]`. According to the reduction relation defined in Sect. 2, the same expression reduces to the partial values `⊥`, `⊥ : ⊥`, `1 : ⊥`, `⊥ : ⊥ : ⊥`, `1 : ⊥ : ⊥`, `⊥ : 2 : ⊥`, `1 : 2 : ⊥`, ... If operations are non-terminating, it is necessary to consider partial result values in general. For instance, `ints1 0` and `ints2 0` do not evaluate to a value but they evaluate to the different partial values `0 : 1 : ⊥` and `0 : 2 : ⊥`, respectively, which shows the non-equivalence of `ints1` and `ints2` by Corollary 1. Thus, one may wonder whether for “well behaved” operations it suffices to consider only result *values*. This good behavior is captured by the property that a function returns a value for any argument value, see Definition 3. Unfortunately, the answer is negative.

Definition 3 (Terminating, totally defined). *Let f be an operation of type $\tau \rightarrow \tau'$. f is terminating if, for all values t of type τ , any rewrite sequence $f t \rightarrow t_1 \rightarrow t_2 \rightarrow \dots$ is finite. f is totally defined if, for all values t of type τ , rewrite rule `Fun` is applicable to $f t$.*

Requiring termination as a condition of good behavior is necessary, as the operations `ints1` and `ints2` show. Total definedness is also necessary, as can be seen by this example:

```

g1 x = 1 : head []
g2 x = 2 : head []

```

`g1` and `g2` are terminating but `head` is not totally defined. Actually, both `g1 0` and `g2 0` have no value but they are not equivalent: `head (g1 0)` and `head (g2 0)` evaluate to 1 and 2, respectively.

Example 5. Functions `h1` and `h2`, defined below, are totally defined and terminating. For any Boolean value t , `h1 t` and `h2 t` produce the same value result, namely t . However, `h1` and `h2` are not observably equivalent when applied to argument `failed` as witnessed by `g`.

```

h1 True  = Just True
h1 False = Just False
h2 x     = Just x
g (Just _) = 0

```

⁴ A possible definition is: `failed = head []`.

Note that we have to use partial input values for equivalence tests even if all relevant operations are terminating and totally defined. This has been shown in Example 4, since both operations of this example are terminating and totally defined.

Now we have enough refined criteria to implement an equivalence checker with a property-based checking tool.

5 Property-Based Checking

Property-based testing is a useful technique to obtain reliable software systems. Testing cannot verify the correctness of programs, but it can be performed automatically and it might prevent wasting time when attempting to prove incorrect properties. If proof obligations are expressed as properties, i.e., Boolean expressions parameterized over input data, and we test these properties with a lot of input data, we have a higher confidence in the correctness of the properties. This motivates the use of property testing tools which automate the checking of properties by random or systematic generation of test inputs. Property-based testing has been introduced with the QuickCheck tool [8] for the functional language Haskell and adapted to other languages, like PrologCheck [1] for Prolog, PropEr [22] for the concurrent functional language Erlang, and EasyCheck [7] and CurryCheck [15] for the functional logic language Curry. If the test data is generated in a systematic (and not random) manner, like in SmallCheck [26], GAST [20], EasyCheck [7], or CurryCheck [15], these tools can actually verify properties for finite input domains. In the following, we show how to extend the property-based test tool CurryCheck to support equivalence checking of operations.

Properties can be defined in source programs as top-level entities with result type `Prop` and an arbitrary number of parameters. CurryCheck offers a predefined set of property combinators to define properties. In order to compare expressions involving non-deterministic operations, CurryCheck offers the property “`<~>`” which has the type `a → a → Prop`. It is satisfied if both arguments have identical result sets. For instance, we can state the requirement that permutations do not change the list length by the property

```
permLength xs = length (perm xs) <~> length xs
```

Since the left argument of “`<~>`” evaluates to many (expectedly identical) values, it is relevant that “`<~>`” compares result *sets* (rather than multi-sets). This is reasonable from a declarative programming point of view, since it is irrelevant how often some result is computed.

Corollary 1 provides a specific criterion for equivalence testing: Two operations `f1` and `f2` are equivalent if, for any partial argument value, they produce the same partial result value. Since partial values cannot be directly compared, we model partial values by extending total values with an explicit \perp constructor. For instance, consider the data types used in Sect. 1. Assume that they are defined by

```
data AB = A | B
data C  = C AB
```

We define their extension to partial values by renaming all constructors and adding a \perp constructor to each type:

```
data P_AB = Bot_AB | P_A | P_B
data P_C  = Bot_C  | P_C P_AB
```

In order to compare the partial results of two operations, we introduce operations that return the partial value of an expression w.r.t. a given partial value, i.e., the expression is partially evaluated up to the degree required by the partial value (and it fails if the expression has not this value). These operations can easily be implemented for each data type:

```
peval_AB :: AB -> P_AB -> P_AB
peval_AB _ Bot_AB = Bot_AB           -- no evaluation
peval_AB A P_A    = P_A
peval_AB B P_B    = P_B

peval_C :: C -> P_C -> P_C
peval_C _ Bot_C  = Bot_C           -- no evaluation
peval_C (C x) (P_C y) = P_C (peval_AB x y)
```

Now we can test the equivalence of `f` and `g` by evaluating both operations to the same partial value. Thus, a single test consists of the application of each operation to an input `x` and a partial result value `p` together with checking whether these applications produce `p`:

```
f_equiv_g x p = peval_C (f x) p <~> peval_C (g x) p
```

To check this property, `CurryCheck` systematically enumerates partial values for `x` (see below how this can be implemented) and values for `p`. During this process, `CurryCheck` generates the inputs `x = failed` and `p = (P_C Bot_AB)` for which the property does not hold. This shows that `f` and `g` are not equivalent.

In a similar way, we can model partial list result values and test whether `sort` and `sort'`, as defined in Sect. 3, are equivalent. If the domain of list elements has three values (like the standard type `Ordering` with values `LT`, `EQ`, and `GT`), `CurryCheck` reports a counter-example (a list with three different elements computed up to the first element) with the 89th test. The high number of tests is due to the fact that test inputs as well as partial output values are enumerated to test each property.

The number of test cases can be significantly reduced by a different encoding. Instead of enumerating operation inputs as well as partial result values, we can enumerate operation inputs only and use a non-deterministic operation which returns *all* partial result values of some given expression. For our example types, these operations can be defined as follows:

```
pval0f_AB :: AB -> P_AB
pval0f_AB _ = Bot_AB
pval0f_AB A = P_A
pval0f_AB B = P_B

pval0f_C :: C -> P_C
pval0f_C _ = Bot_C
pval0f_C (C x) = P_C (pval0f_AB x)
```

Now we can test the equivalence of f and g by checking whether both operations have the same set of partial values for a given input:

```
f_equiv_g x = pval0f_C (f x) <~> pval0f_C (g x)
```

CurryCheck returns the same counter-example as before. This is also true for the permutation sort example, but now the counter-example is found with the 11th test.

Due to the reduced search space of our second implementation of equivalence checking, we might think that this method should always be preferred. However, in case of non-terminating operations, it is less powerful. For instance, consider the operations `ints1` and `ints2` of Example 2. Since `ints1 0` has an infinite set of partial result values, the equivalence test with `pval0f` operations would try to compare sets with infinitely many values. Thus, it would not terminate and does not yield a counter-example. However, the equivalence test with `peval` operations returns a counter-example by fixing a partial term (e.g., a partial list with at least two elements) and evaluating `ints1` and `ints2` up to this partial list.

Based on these considerations, equivalence checking is implemented in CurryCheck as follows. First, CurryCheck provides a specific “operation equivalence” property denoted by `<=>`. Hence,

```
f_equiv_g = f <=> g
```

denotes the property that f and g are equivalent operations. In contrast to other properties like “`<~>`”, which are implemented by some Curry code [7], the property “`<=>`” is just a marker⁵ which will be transformed by CurryCheck into a standard property based on the results of Sect. 4. For this purpose, CurryCheck transforms the property above as follows:

1. If both operations f and g are terminating, then the sets of partial result values are finite so that these sets can be compared in a finite amount of time. Thus, if T is the result type of f and g , the auxiliary operation `pval0f_T` (and similarly for all types on which T depends) is generated as shown above and the following property is generated:

```
f_equiv_g x = pval0f_T (f x) <~> pval0f_T (g x)
```

2. Otherwise, for each partial value, CurryCheck tests whether both operations compute this result. Thus, if T is the result type of f and g , the auxiliary operation `peval_T` (and similarly for all types on which T depends) is generated as shown above and the following property is generated:

```
f_equiv_g x p = peval_T (f x) p <~> peval_T (g x) p
```

In order to decide between these transformation options, CurryCheck uses the analysis framework CASS [17] to approximate the termination behavior of both operations. If the termination property of both operations can be proved (for this purpose, CASS uses an ordering on arguments in recursive calls), the first transformation is used, otherwise the second. If the termination cannot be proved

⁵ CurryCheck also ensures that both arguments of “`<=>`” are defined operations, otherwise an error is reported.

but the programmer is sure about the termination of both operations, he can also mark the property with the suffix `'TERMINATE` to tell CurryCheck to use the first transformation.

Example 6. Consider the recursive and non-recursive definition of the McCarthy 91 function:

```
mc91r n = if n > 100 then n-10 else mc91r (mc91r (n+11))
mc91n n = if n > 100 then n-10 else 91
```

Since CASS is not able to check the termination of `mc91r`, we annotate the equivalence property so that CurryCheck uses the first transformation:

```
mc91r_equiv_mc91n'TERMINATE = mc91r <=> mc91n
```

Due to the results of Sect. 4, the generated properties must be checked with all *partial* input values. In the default mode, CurryCheck generates (total) values for input parameters of properties. However, CurryCheck also supports the definition of user-defined generators for input parameters (see [15] for details). For instance, one can define a generator for partial Boolean values by

```
genBool = genCons0 failed ||| genCons0 False ||| genCons0 True
```

CurryCheck automatically defines generators for partial values of all data types occurring in equivalence properties.

According to the results of Sect. 4, checking the above properties allows us to find counter-examples for non-equivalent operations if the domain of values is finite (as in the example of Sect. 1) or we enumerate enough test inputs. An exception are specific non-terminating operations. For instance, consider the contrived operations

```
k1 = [loop, True]
k2 = [loop, False]
```

where the evaluation of `loop` does not terminate. The non-equivalence of `k1` and `k2` can be detected by evaluating them to `[⊥, True]` and `[⊥, False]`, respectively. Since a systematic enumeration of all partial values might generate the value `[True, ⊥]` before `[⊥, True]`, CurryCheck might not find the counter-example due to the non-termination of `loop` (since CurryCheck performs all tests in a sequential manner). Fortunately, this is a problem which rarely occurs in practice. Not all non-terminating operations are affected by this problem but only operations that loop without producing any data. For instance, the non-equivalence of `ints1` and `ints2` of Example 2 can be shown with our approach. Such operations are called *productive* in [16]. Intuitively, productive operations always generate some data after a finite number of steps.

In order to avoid such non-termination problems when CurryCheck is used in an automatic manner (e.g., by a software package manager), CurryCheck has an option for a “safe” execution mode. In this mode, operations involved in an equivalence property are analyzed for their productivity behavior. If it cannot be proved that an operation is productive (by approximating their run-time behavior with CASS), the equivalence check for this operation is ignored. This ensures the termination of all equivalence tests. The restriction to productive

operations is not a serious limitation since, as evaluated in [16], most operations occurring in practical programs are actually productive. If there are operations where CurryCheck cannot prove productivity but the programmer is sure about this property, the property can be annotated with the suffix 'PRODUCTIVE so that it is also checked in the safe mode.

Example 7. Consider the definition of all prime numbers by the sieve of Eratosthenes:

```
primes = sieve [2..]
  where sieve (x:xs) = x : sieve (filter (\y -> y `mod` x > 0) xs)
```

After looking at the first four values of this list, a naive programmer might think that the following prime generator is much simpler:

```
dummy_primes = 2 : [3,5..]
```

Testing the equivalence of these two operations is not possible in the safe mode, since the productivity of `primes` depends on the fact that there are infinitely many prime numbers. Hence, a more experienced programmer would annotate the equivalence test as

```
primes_equiv'PRODUCTIVE = primes <=> dummy_primes
```

so that it will be tested even in the safe mode and CurryCheck finds a counterexample (evaluating the result list up to the first five elements) to this property.

6 Related Work

Equivalence of operations was defined for functional logic programs in [5]. There, this notion is applied to relate specifications and implementations. Moreover, it is shown how to use specifications as dynamic contracts to check the correct behavior of implementations at run-time, but static methods to check equivalence are not discussed.

Bacci et al. [6] formalized various notions of equivalence, as reviewed in Sect. 1, and developed the tool AbsSpec which derives specifications, i.e., equations up to some fixed depth of the involved expressions, from a given Curry program. Although the derived specifications are equivalent to the implementation, their method cannot be used to check the equivalence of arbitrary operations (and AbsSpec does no longer work at the time of this writing).

QuickSpec [9] has similar goals as AbsSpec but is based on a different setting. QuickSpec infers specifications in form of equations from a given functional program but it uses a black box approach, i.e., it uses testing to infer program properties. Thus, it can be seen as an intermediate approach between AbsSpec and our approach: similarly to our approach, QuickSpec uses property-based testing to check the correctness of specifications, but it is restricted to functional programs, which simplifies the notion of equivalence.

Our method to check equality of computed results for all partial values is also related to test properties in non-strict functional languages [10]. Thanks to the non-deterministic features of Curry, our approach does not require impure

features like `isBottom` or `unsafePerformIO`, which are used in [10] to compare partial values.

Partial values as inputs for property-based testing are also used in `Lazy SmallCheck` [26], a test tool for Haskell which generates data in a systematic (not random) manner. Partial input values are used to reduce the number of test cases: if a property is satisfied for a partial value, it is also satisfied for all refinements of this partial value so that it is not necessary to test these refinements. Thus, `Lazy SmallCheck` exploits partial values to reduce the number of test cases for total values, where in our approach partial values are used to avoid testing with all possible contexts and to find counter examples which might not be detected with total values only. In contrast to our explicit encoding of partial values, which is possible due to the logic features of Curry, `Lazy SmallCheck` represents partial values as run-time errors which are observed using imprecise exceptions [24].

The use of property-based testing to check the equivalence of operations in a software package manager with support for semantic versioning is proposed in [16]. This approach concentrates on ensuring the termination of equivalence checking by introducing the notion of productive operations. However, for terminating operations only ground equivalence is tested so that the proposed semantic versioning checking method is more restricted than in our case. The results presented in this paper can be used to generalize this semantic versioning tool.

7 Conclusions

We have presented a method to check the equivalence of operations defined by a functional logic program. This method is useful for software package managers to provide automatic semantic versioning checks, i.e., to compare two different versions of a software package, or to check the correctness of an implementation against a specification. Since we developed our results for a non-strict functional logic language, the same techniques can be used to test equivalence in purely functional languages, e.g., for Haskell programs.

We have shown that the general equivalence of operations, which requires that the same values are computed in all possible contexts, can be reduced to checking or proving equality of partial results terms. Our results support the use of automatic property-based test tools for equivalence checking. Although this method is incomplete, i.e., it does not formally ensure equivalence, it provides a high confidence and prevent wasting time in attempts to prove incorrect equivalence properties. Moreover, the presented results could also be helpful for manual proof construction or using proof assistants.

For future work, we will integrate our method in the software package manager CPM [16]. Furthermore, it is interesting to explore how automatic theorem provers can be used to verify specific equivalence properties.

Acknowledgments. The authors are grateful to Finn Teegen for constructive remarks to an initial version of this paper, and to the anonymous reviewers for their helpful

comments to improve this paper. This material is based in part upon work supported by the National Science Foundation under Grant No. 1317249.

References

1. Amaral, C., Florido, M., Santos Costa, V.: PrologCheck – property-based testing in Prolog. In: Codish, M., Sumii, E. (eds.) FLOPS 2014. LNCS, vol. 8475, pp. 1–17. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07151-0_1
2. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. *J. ACM* **47**(4), 776–822 (2000)
3. Antoy, S., Hanus, M.: Overlapping rules and logic variables in functional logic programs. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 87–101. Springer, Heidelberg (2006). https://doi.org/10.1007/11799573_9
4. Antoy, S., Hanus, M.: Functional logic programming. *Commun. ACM* **53**(4), 74–85 (2010)
5. Antoy, S., Hanus, M.: Contracts and specifications for functional logic programming. In: Russo, C., Zhou, N.-F. (eds.) PADL 2012. LNCS, vol. 7149, pp. 33–47. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27694-1_4
6. Bacci, G., Comini, M., Feliú, M.A., Villanueva, A.: Automatic synthesis of specifications for first order Curry. In: Principles and Practice of Declarative Programming (PPDP 2012), pp. 25–34. ACM Press (2012)
7. Christiansen, J., Fischer, S.: EasyCheck — test data for free. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 322–336. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78969-7_23
8. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: International Conference on Functional Programming (ICFP 2000), pp. 268–279. ACM Press (2000)
9. Claessen, K., Smallbone, N., Hughes, J.: QUICKSPEC: guessing formal specifications using testing. In: Fraser, G., Gargantini, A. (eds.) TAP 2010. LNCS, vol. 6143, pp. 6–21. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13977-2_3
10. Danielsson, N.A., Jansson, P.: Chasing bottoms: a case study in program verification in the presence of partial and infinite values. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 85–109. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27764-4_6
11. Echahed, R., Janodet, J.-C.: On constructor-based graph rewriting systems. Research report IMAG 985-I, IMAG-LSR, CNRS, Grenoble (1997)
12. Echahed, R., Janodet, J.-C.: Admissible graph rewriting and narrowing. In: Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP 1998), pp. 325–340 (1998)
13. González-Moreno, J.C., Hortalá-González, M.T., López-Fraguas, F.J., Rodríguez-Artalejo, M.: An approach to declarative programming based on a rewriting logic. *J. Logic Program.* **40**, 47–87 (1999)
14. Hanus, M.: Functional logic programming: from theory to Curry. In: Voronkov, A., Weidenbach, C. (eds.) Programming Logics. LNCS, vol. 7797, pp. 123–168. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37651-1_6
15. Hanus, M.: CurryCheck: checking properties of curry programs. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 222–239. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63139-4_13

16. Hanus, M.: Semantic versioning checking in a declarative package manager. In: Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017). OpenAccess Series in Informatics (OASICS), pp. 6:1–6:16 (2017)
17. Hanus, M., Skrlac, F.: A modular and generic analysis server system for functional logic programs. In: Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM 2014), pp. 181–188. ACM Press (2014)
18. Hanus, M. (ed.): Curry: an integrated functional logic language (vers. 0.9.0) (2016). <http://www.curry-language.org>
19. Hussmann, H.: Nondeterministic algebraic specifications and nonconfluent term rewriting. *J. Logic Program.* **12**, 237–255 (1992)
20. Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: GAST: generic automated software testing. In: Peña, R., Arts, T. (eds.) IFL 2002. LNCS, vol. 2670, pp. 84–100. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44854-3_6
21. López-Fraguas, F.J., Rodríguez-Hortalá, J., Sánchez-Hernández, J.: A simple rewrite notion for call-time choice semantics. In: Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2007), pp. 197–208. ACM Press (2007)
22. Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: Proceedings of the 10th ACM SIGPLAN Workshop on Erlang, pp. 39–50 (2011)
23. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries—The Revised Report. Cambridge University Press, Cambridge (2003)
24. Peyton Jones, S., Reid, A., Henderson, F., Hoare, T., Marlow, S.: A semantics for imprecise exceptions. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI 1999), pp. 25–36. ACM Press (1999)
25. Pitts, A.M.: Operational semantics and program equivalence. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 378–412. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45699-6_8
26. Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In: Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, pp. 37–48. ACM Press (2008)



Optimizing Declarative Parallel Distributed Graph Processing by Using Constraint Solvers

Akimasa Morihata¹(✉), Kento Emoto², Kiminori Matsuzaki³, Zhenjiang Hu⁴,
and Hideya Iwasaki⁵

¹ University of Tokyo, Tokyo, Japan
morihata@graco.c.u-tokyo.ac.jp

² Kyushu Institute of Technology, Kitakyushu, Japan

³ Kochi University of Technology, Kami, Japan

⁴ National Institute of Informatics, Tokyo, Japan

⁵ The University of Electro-Communications, Chofu, Japan

Abstract. Vertex-centric graph processing is a promising approach for facilitating development of parallel distributed graph processing programs. Each vertex is regarded as a tiny thread and graph processing is described as cooperation among vertices. This approach resolves many issues in parallel distributed processing such as synchronization and load balancing. However, it is still difficult to develop efficient programs requiring careful problem-specific tuning. We present a method for automatically optimizing vertex-centric graph processing programs. The key is the use of constraint solvers to analyze the subtle properties of the programs. We focus on a functional vertex-centric graph processing language, Fregel, and show that quantifier elimination and SMT (Satisfiability Modulo Theories) are useful for optimizing Fregel programs. A preliminary experiment indicated that a modern SMT solver can perform optimization within a realistic time frame and that our method can significantly improve the performance of naively written declarative vertex-centric graph processing programs.

1 Introduction

Nowadays big graphs are ubiquitous. Nearly every interesting data set, such as those for customer purchase histories, social networks, and protein interaction networks, consists of big graphs. Parallel distributed processing is necessary for analyzing big graphs that cannot fit in the memory of a single machine. However, parallel distributed processing is difficult due to such issues as communications, synchronizations, and load balancing.

Vertex-centric graph processing (abbreviated to VcGP) [1] is a promising approach for reducing the difficulties of parallel distributed graph processing. VcGP is based on the “think like a vertex” programming style. It regards each vertex as a tiny thread and describes graph processing as cooperation among vertices, each of which updates its value using information supplied from other vertices.

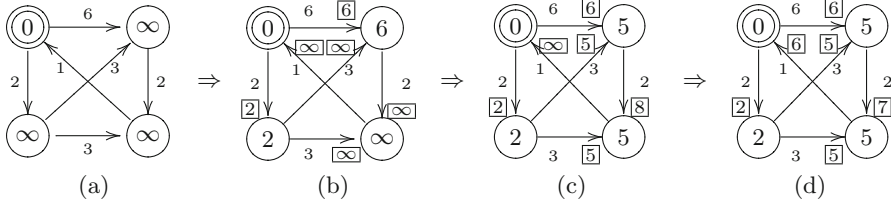


Fig. 1. Vertex-centric SSSP: doubly-circled vertex denotes source, and framed numbers denote messages.

As an example, consider the following algorithm illustrated in Fig. 1 for the single-source shortest path problem (SSSP).

- First, the source vertex is assigned 0, and the other vertices are assigned ∞ . The values are the estimated distances from the source vertex (Fig. 1 (a)).
- Then, each vertex sends the estimated distances to its neighbors and updates its value if it receives a shorter distance (Fig. 1(b–c)).
- The previous step is repeated until all the values are no longer changed (Fig 1(d)).

A VcGP framework executes programs in a distributed environment. Vertices (and accordingly edges) are distributed among computational nodes. At every step, every computational node simultaneously updates the values of its vertices in accordance with the specified computation. Computational nodes exchange messages if information of other nodes is necessary. The VcGP approach releases programmers from such typical difficulties as communication, synchronization, and load balancing, and makes it easier to write *runnable* parallel distributed graph processing programs.

Although the VcGP approach is beneficial, it is still difficult to achieve *efficiency*. Natural VcGP programs tend to be slow. For instance, there is room for improvement in the above SSSP algorithm.

- It is unnecessary to process all vertices at every step; it is sufficient to process only those vertices for which values are updated. Similarly, it is unnecessary for all vertices to communicate their neighbors at every step.
- We have adopted *synchronous* execution: each vertex is processed exactly once at each step. One can instead adopt *asynchronous* execution, which processes vertices without synchronization barriers. The relative efficiencies of these two approaches depend on the situation. For SSSP, both executions lead to the same solution, and combining the two approaches may improve performance.
- The algorithm is essentially the Bellman-Ford algorithm, whose work is $O(n^2)$, where n is the size of the graph. Processing near-source vertices prior to distant ones, like Dijkstra’s algorithm, may reduce the amount of work because the work of Dijkstra’s algorithm is $O(n \log n)$.

These inefficiencies have already been known, and several frameworks have been proposed to remove them [1–7]. For instance, the Pregel framework [1] enables

us to *inactivate* vertices so that they are ignored by the runtime system until they receive a new message. However, it remains the programmer’s responsibility to be aware of these inefficiencies and to mitigate them by using available functionalities. This is fairly difficult because these inefficiencies and potential improvements may be hidden by nontrivial problem-specific properties.

We have developed a method for automatically removing such inefficiencies from naive programs written in a functional VcGP language, Fregel [8]. The key is the use of modern constraint solvers for identifying potential optimizations. The declarative nature of Fregel enables optimizations to be directly reduced to constraint solving problems. We focused on four optimizations.

- Eliminating redundant communications (Sect. 3.1).
- Inactivating vertices that do not need to be processed (Sect. 3.2).
- Removing synchronization barriers and thereby enabling asynchronous execution (Sect. 3.3).
- Introducing priorities for processing vertices (Sect. 3.4).

Our approach is not specific to these optimizations for Fregel programs. Nontrivial optimizations on declarative VcGP languages will be implemented similarly if they are formalized as constraint solving problems.

We considered the use of two different constraint solving methods: quantifier elimination (QE) [9] and satisfiability modulo theories (SMT) [10]. The former enables the use of arbitrary quantifier nesting and can generate the program fragments that are necessary for the optimizations; therefore, it is suitable for formalizing optimizations. However, it is less practical because of its high computational cost. We thus use SMT solvers as a practical implementation method that captures typical cases. An experiment using a proof-of-concept implementation demonstrated that a modern SMT solver can perform the optimizations within a realistic time frame and that the optimizations led to significant performance improvement (Sect. 4).

2 Fregel: Functional VcGP Language

2.1 Pregel

Pregel [1] is a pioneering VcGP framework. We review it first because the following systems were strongly influenced.

A Pregel program essentially consists of a function that is invoked by each vertex at each step. It usually updates the values stored in vertices using the following functionalities.

- A vertex can send messages usually to adjacent vertices. The message is available on the destination at the next step.
- A vertex can inactivate itself. The runtime system skips processing of inactive vertices. An inactive vertex is reactivated if it receives a message.
- A vertex can read a summary showing the total sum, average, etc. of all active vertices. This functionality is called an *aggregator*.

```

sssp g = let init v = if v is the source vertex then 0 else ∞
          step v prev = let m = minimum [prev u + e | (e,u) <- is v]
                        in min (prev v) m
          in fregel init step Fix g

```

Fig. 2. Fregel pseudo-program for single-source shortest path problem.

Pregel is based on the bulk synchronous parallel (BSP) model [11]. Computations on the BSP model consists of a series of supersteps. A superstep is a local computation (i.e., invocation of the function by each vertex) followed by a synchronization barrier that guarantees message arrival. Because of the adoption of the BSP model, computation of a Pregel program is deterministic¹: every vertex can be processed simultaneously without any race conditions. A Pregel program terminates when all vertices become inactive.

2.2 Fregel

Fregel [8] is a functional VcGP language. It is a subset of Haskell and enables VcGP programs to be easily written using graph-processing higher-order functions that conceal side effects including communication and vertex inactivation.

Figure 2 shows a pseudo-program for SSSP. For readability, we focus on the core functionality of Fregel and accordingly use a simplified syntax.

The core of Fregel is the graph processing higher-order function `fregel`. Its first parameter, `init :: Vertex -> Int` in Fig. 2, is applied to each vertex at the initial step. The second one, `step :: Vertex -> (Vertex -> Int) -> Int`, is used at the subsequent steps. The function `step` takes vertex `v` and a table, `prev :: Vertex -> Int`, which stores the results of the previous step. A vertex may access results of neighbor vertices using the table and a special function called a *generator*. The program in Fig. 2 uses `is :: Vertex -> [(Edge, Vertex)]`, which enumerates every neighbor with an edge that leads to the neighbor. Other generators can express other communication patterns including aggregators. Since information read from the neighbors essentially forms a multiset, the information should be summarized not by using a conventional list operation but by using an associative commutative binary operation such as `sum` and `minimum`. The operation should have the unit needed for dealing with an isolated vertex. The third parameter of `fregel`, namely `Fix`, shows that the computation terminates when the result of the current step is the same as that of the previous one. Note that Fregel has no construct for inactivating vertices.

The Fregel compiler translates a Fregel program into a Java program runnable on Giraph². The functionalities of Giraph are nearly the same as those of Pregel. An access to a neighbor's previous value using the `prev` table and a generator is compiled to message exchange if the target vertex is located in a different computational node. Calculating new vertex values using neighbors' previous values naturally corresponds to a superstep in the BSP model.

¹ Except for the order of arrival messages.

² Apache Giraph: <http://giraph.apache.org/>.

$$\begin{aligned}
\text{step } v \text{ prev} = & \text{let } c_1 = \bigoplus_1 [f_1(e, \text{prev } u) \mid (e, u) \leftarrow \text{is } v, p_1(\text{prev } u)] \\
& \vdots \\
& c_n = \bigoplus_n [f_n(e, \text{prev } u) \mid (e, u) \leftarrow \text{is } v, p_n(\text{prev } u)] \\
\text{in } & g(\text{prev } v, c_1, \dots, c_m)
\end{aligned}$$

Fig. 3. Target program for optimization

3 Optimizing Fregel Programs

Here we describe optimizations for programs written using a `fregel` function. We refer to the second parameter, i.e., the one invoked at the non-initial steps, as `step`, and assume that it is written in the form shown in Fig. 3. In the program, f_i , p_i , and \bigoplus_i ($1 \leq i \leq n$) respectively represent computation over each neighbor’s information, the condition showing the necessity of sending the information, and the operator used to summarize the information. g denotes the calculation of the new value of the vertex. For simplicity, we assume the termination condition is `Fix` and only the `is` function is used as a generator. We discuss these limitations in Sect. 3.5.

We use SSSP as a running example. For SSSP in Fig. 2, $n = 1$, $f_1 = (+)$, $g = (\bigoplus_1) = \min$, and $p_1(x) = \text{True}$.

3.1 Reducing Communication

Since accesses of a neighbor’s information are compiled to message exchange, modifying the condition p_i and thereby avoiding unnecessary accesses reduces the amount of communications. In the following discussion, we focus on reducing communications caused by the k -th access expressed by f_k , p_k , and \bigoplus_k . Our strategy is to formalize the situation in which optimization is possible and then to use constraint solvers to implement the optimization.

Formulation. Let \dot{u} be the value of the message-sending vertex, and consider formulating the necessity of sending \dot{u} . The following property naturally formulates that sending \dot{u} does not affect the computation on the destination vertex.

$$\begin{aligned}
& \forall v, e, w_1, \dots, w_n. \\
& g(v, w_1, \dots, w_n) = g(v, w_1, \dots, w_{k-1}, w_k \oplus_k f_k(e, \dot{u}), w_{k+1}, \dots, w_n) \quad (1)
\end{aligned}$$

Though correct, this property is not sufficient in practice, as the following example shows.

Example: SSSP. For SSSP, Property (1) is instantiated as

$$\forall v, e, w. \min(v, w) = \min(v, \min(w, e + \dot{u})).$$

This is equivalent to $\dot{u} = \infty$, which means that a vertex can skip message sending if its value is ∞ . This result is not satisfactory because a vertex can skip message sending if its value is unchanged from the previous step.

For capturing the case of SSSP, we need a more general formulation that takes the previous value into account. A vertex may be able to skip message sending if sufficient information had been sent at the previous step. The following formula captures the idea. Here, \dot{u} and \ddot{u} respectively denote the current and previous values of the vertex.

$$\begin{aligned} &\forall v, e, w_1, \dots, w_n, w'_1, \dots, w'_n. \\ &g(v', w'_1, \dots, w'_n) = g(v', w'_1, \dots, w'_k \oplus_k f_k(e, \dot{u}), w'_{k+1} \dots, w'_n) \\ &\quad \mathbf{where} \quad v' = g(v, w_1, \dots, w_k \oplus_k f_k(e, \ddot{u}), w_{k+1} \dots, w_n) \end{aligned} \quad (2)$$

This is a generalization of Property (1). The necessity of \dot{u} is checked on the basis of the premise that the message-receiving vertex (which has value v') took into account the previous value \ddot{u} of the message-sending vertex.

Example: SSSP (Contd). Property (2) is instantiated as

$$\begin{aligned} &\forall v, e, w, w'. \min(v', w') = \min(v', \min(w', e + \dot{u})) \\ &\quad \mathbf{where} \quad v' = \min(v, \min(w, e + \ddot{u})). \end{aligned}$$

This is equivalent to $\dot{u} \geq \ddot{u}$: a vertex can skip communication when the current value is not smaller than the previous one. Since the current value is always not larger than the previous one, this is equivalent to $\dot{u} = \ddot{u}$.

Implementation Using Constraint Solvers. We could implement the optimization by checking Property (2) dynamically for each vertex. However, since Property (2) consists of quantifiers, its evaluation is likely impossible or very slow. To obtain efficient codes, we need a method for *synthesizing a simple, especially quantifier-free, formula* that is equivalent to (or expressing a sufficient condition of) the property. For this purpose, we use constraint solvers.

QE translates a formula into a quantifier-free equivalent. For example, it may translate $\forall x. x^2 + ax + b \geq 0$ into $4b - a^2 \geq 0$. While QE is theoretically ideal for our purpose, there are three reasons that using QE solvers may be impractical. First, there are only a few formal systems for which QE procedures are known. Second, QE procedures are usually very slow. Third, current implementations of QE tend to be experimental. Nevertheless, it is worthwhile to formulate the optimizations as QE because these problems may one day be solved.

As a more practical implementation, we propose using SMT instead of QE. Given a closed formula consisting of only one kind of quantifier, SMT checks (i.e., does not translate) whether it is satisfiable. For example, it may answer “yes” for $\forall x, a. x^2 + ax + a^2 \geq 0$. Recently efficient SMT solvers are intensively developed and used in many applications.

There are two problems in using SMT for checking Property (2). The property contains free variables, \dot{u} and \ddot{u} , and moreover, SMT solvers are unable to

```

sssp g =
  let init v = (if v is the source vertex then 0 else ∞, False)
      step v prev = let m = minimum [fst (prev u) + e |
                                   (e,u) <- is v, not (snd (prev u))]
                    v' = min (fst (prev v)) m
                    in (v', v' == fst (prev v))
  in fregel init step Fix g

```

Fig. 4. Fregel SSSP program obtained by communication reduction, where `fst` and `snd` respectively denote extraction of the first and second components from a pair.

synthesize a simple formula. To overcome these problems, we prepare templates of simple formulae, such as $\dot{u} = \ddot{u}$. If the SMT solver guarantees that a template is a sufficient condition of Property (2), we insert the negation of the template into p_k . The effectiveness of this approach relies on the generality of the template. We believe $\dot{u} = \ddot{u}$ captures most practical cases. Other useful templates include natural comparisons on \dot{u} and \ddot{u} , such as \geq and/or \leq on numbers and lexicographic orders on tuples.

Example: SSSP. We instruct an SMT solver to check the following formula.

$$\forall \dot{u}, \ddot{u}, v, e, w, w'. (\dot{u} = \ddot{u}) \Rightarrow (\min(v', w') = \min(v', \min(w', e + \dot{u})))$$

where $v' = \min(v, \min(w, e + \ddot{u}))$

The solver verifies the condition. We thus modify the program as follows. We instruct each vertex to check and remember the truth of the template; then, we modify p_1 so that it checks the remembered truth. Figure 4 shows the optimized program. Each node value is a pair, $\dot{u} = (d, b)$, where d and b respectively denote the estimated distance from the source and whether the value has been changed.

3.2 Inactivating Vertices

Next we discuss inactivating vertices. Inactive vertices do nothing (including any message sending) unless they receive a new message. This optimization should be applied after communication reduction optimization described in Sect. 3.1 because we cannot inactivate vertices that send a message.

A vertex is inactivated if the following condition holds: unless the vertex receives a message, its value does not change and it does not need to send a message. The optimization condition is thus formalized as

$$\left(\bigwedge_{1 \leq i \leq n} \neg p_i(\dot{u}) \right) \wedge (g(\dot{u}, \iota_1, \dots, \iota_n) = \dot{u}), \quad (3)$$

where each ι_i ($1 \leq i \leq n$) is the unit of \oplus_i and corresponds to the absence of messages. Since Property 3 contains no quantifier, this optimization can be implemented without the use of a constraint solver.

Example: SSSP. For the SSSP in Fig. 4, Property (3) is instantiated to $b \wedge (\min(d, \infty) = d)$ **where** $\dot{u} = (d, b)$, which is equivalent to b **where** $\dot{u} = (d, b)$. In short, a vertex can be inactivated if its value is the same as the previous one.

3.3 Removing Barriers

Recall that the execution of Fregel is based on the BSP model. Each local computation is followed by a synchronization barrier. Though this makes program behaviors deterministic and more understandable, barriers may make execution slower especially when there are many computational nodes. For most graph algorithms including SSSP, asynchronous barrier-less execution and synchronous barrier-full execution yield the same result; thus, barriers are unnecessary.

The flexibility of asynchronous execution enables further optimizations such as vertex splitting (also known as vertex mirroring) [12, 13]. Practical graphs often contain vertices that have too many edges, and such vertices form a bottleneck in VcGP. Vertex splitting resolves the bottleneck by splitting these vertices and distributing their edges among the computational nodes. With synchronous execution, vertex splitting requires an additional phase for every step to merge the messages sent to the split vertices. With asynchronous execution, the additional phase is unnecessary because message delay does not matter. Another possible optimization is to repeatedly process vertices in the same computational node before sending messages to other nodes. This optimization is related to subgraph-centric (or neighborhood-centric) approaches [4, 5] in which not vertices but subgraphs are the target of parallel processing.

We have developed a method that automatically guarantees equivalence between synchronous and asynchronous execution. We first present the following lemma. Its proof is obvious and thus omitted.

Lemma 1. *For functions h , h' and a binary relation \preceq , three conditions are assumed:*

Monotonicity of h : $\forall x, y. (x \preceq y) \Rightarrow (h(x) \preceq h(y))$.

Ordering of h and h' : $\forall x. (x \preceq h'(x)) \wedge (h'(x) \preceq h(x))$.

Antisymmetry of \preceq : $\forall x, y. (x \preceq y \wedge y \preceq x) \Rightarrow (x = y)$.

Then, $h^(x) = h^*(h'(x))$ holds for any x , where h^* is defined by $h^*(x) = z \iff (h(z) = z) \wedge (z = h(h(\dots(h(x))\dots))$. \square*

We apply Lemma 1 as follows. We regard h as a complete one-step processing of the graph. Similarly, we regard h' as a partial processing in which some vertices and messages are skipped. We regard asynchronous execution as a series of partial processing. Lemma 1 guarantees that a partial processing does not change the result; then, by induction, asynchronous execution does not as well.

Lemma 1 requires an appropriate binary relation, \preceq . From the ordering between h and h' , a natural candidate is the comparison of the progress in computation: $g_1 \preceq g_2$ indicates that graph g_2 can be obtained by processing computation from g_1 . Another issue for using Lemma 1 is the gap between

graph processing and vertex processing. While h , h' , and \preceq deal with graphs, we would like to consider vertex-processing functions. The following lemma bridges the gap. For simplicity, we assume that the **step** function contains only one access of neighbor's information.

Lemma 2. *For the **step** function, let \preceq be a binary relation defined by $x \preceq y \iff (\exists m. y = g(x, m))$. Three conditions are assumed:*

- $\forall x, m, m'. g(x, m \oplus m') = g(g(x, m), m')$.
- $\forall x, y. (x \preceq y \wedge y \preceq x) \Rightarrow (x = y)$.
- $\forall x, y, z. (x \preceq y) \Rightarrow (g(z, x) \preceq g(z, y))$.

*Then, h_{step} , h'_{step} , and \preceq_G satisfy the premise of Lemma 1, where the first two are respectively complete and partial one-step processing over the graph by **step** and the last one compares graphs based on vertex-wise comparison using \preceq .*

Proof (sketch). The first condition and the definition of \preceq guarantee the ordering between h_{step} and h'_{step} . The antisymmetry of \preceq_G easily follows from the second condition. The third condition together with the first one and the commutativity of \oplus guarantees the monotonicity of h_{step} . \square

The first condition of Lemma 2 can be taken to mean that message delay is not harmful. This is a natural requirement for asynchronous execution.

Example: SSSP. For SSSP, the definition of the relation \preceq is instantiated as $x \preceq y \iff \exists w. \min\{x, w\} = y$, which is equivalent to $x \geq y$. Therefore, confirming the three conditions is easy.

Implementation. The first and second conditions can be checked using either QE or SMT. Note that the second is equivalent to $\forall x, m, w. (g(g(x, m), w) = x) \Rightarrow (g(x, m) = x)$, where y is expressed as $g(x, m)$. Since the definition of \preceq contains an existential quantifier, the third condition cannot be directly checked using SMT. When using an SMT solver, we may instead check the following sufficient condition.

$$\forall x, y, z. (x \preceq y) \Rightarrow (g(g(z, x), y) = g(z, y))$$

This can be read to mean that the old result, x , can be “overwritten” by the newer result, y . This is also natural in asynchronous execution.

3.4 Prioritized Execution

Another interesting optimization asynchronous execution enables is prioritized execution [3, 6, 7]. For example in SSSP, a prioritized execution may more intensively process vertices nearer to the source, like Dijkstra's algorithm.

Prioritized execution typically focuses on vertices whose values are *nearer* to the final outcome and thus likely contribute to the final outcome of other vertices.

Therefore, it is natural to use \preceq defined in Lemma 1, which essentially compares progress in computation, as a priority for processing vertices. For SSSP, \preceq is equivalent to \geq and thus is a perfect candidate.

However, there are two problems with using \preceq for prioritized execution. First, since its definition contains an existential quantifier, it is essentially not executable unless QE is used. The other, more essential problem is that \preceq may not be a linear order. Nonlinear orders cannot be used for processing vertices efficiently using priority queues. A practical solution to this problem is to check whether a known linear order, \geq for example, is consistent with \preceq ; i.e., $\forall x, y. (x \preceq y) \Rightarrow (x \geq y)$. If it is, the linear order can be used for prioritization. Note that an SMT solver can check the condition.

3.5 Limitation and Generalization

We have assumed that information reading from neighbors is expressed using the `is` generator. Use of other kinds of generators, including the one for expressing an aggregator, generally does not introduce any difficulty. We did not assume anything about communication except that the communication topology does not change during computation.

A notable exception is the case of vertex inactivation. Aggregator's result may change regardless of message arrival. Therefore, if the k -th communication is an aggregator, the following condition should be checked instead of Property (3).

$$\left(\bigwedge_{1 \leq i \leq n} \neg p_i(\dot{u}) \right) \wedge (\forall w_k. g(\dot{u}, \iota_1, \dots, w_k, \dots, \iota_n) = \dot{u})$$

Namely, the vertex value should not change regardless of the aggregator's value if the vertex receives no message. Since it contains a quantifier, unless QE is used, an executable sufficient condition is needed. A natural candidate is to check the following condition instead.

$$\forall \dot{u}. \left(\bigwedge_{1 \leq i \leq n} \neg p_i(\dot{u}) \right) \Rightarrow (\forall w_k. g(\dot{u}, \iota_1, \dots, w_k, \dots, \iota_n) = \dot{u})$$

If it holds, a vertex having \dot{u} can be inactivated if $(\bigwedge_{1 \leq i \leq n} \neg p_i(\dot{u}))$ holds. The condition can be checked using SMT.

We have considered only a certain form of programs. For example, termination conditions other than `Fix` and the other graph processing higher-order functions were neglected. The restriction is theoretically not essential. The Fregel compiler normalizes other forms of programs into exactly the one in Fig. 3. Nevertheless, from the practical perspective, since the normalization complicates programs, it is questionable whether normalized programs can be effectively optimized.

4 Implementation and Evaluation

The feasibility of our method was evaluated by implementing it in the Fregel compiler.

Table 1. Graphs for experiments

Name	#Vertices	#Edges
WebBerkStan	685,230	7,600,595
RoadNet-PA	1,088,092	1,541,898
Rand-1M10M	1,000,000	10,000,000

4.1 Implementation

The implementation uses the Z3 SMT solver³ (version 4.3.2). Because the current Fregel backend, Giraph, does not support asynchronous execution, we implemented only communication reduction and vertex inactivation. There is no conceptual difficulty in implementing the other optimizations.

The implementation was mostly straightforward. It may be worth noting that the units for minimum and maximum, respectively $-\infty$ and ∞ , are necessary for vertex inactivation. We prepared numerals with $-\infty$ and ∞ and used them instead of the one conventionally used, such as `Int`.

4.2 Setup of Experiments

We applied our optimizations to three Fregel programs:

- SSSP: the SSSP program shown in Fig. 2.
- PageRank: a program that calculates PageRank by repeatedly calculating a weighted sum of the surrounding vertices' values (30 iterations).
- SCC: a program that calculates strongly connected components by repeatedly finding a strongly connected component by a `fregel` function, which propagates the maximum vertex id, and then removing that component from the graph [14].

For each of them, we considered four programs: the original Fregel program, one to which only the communication reduction was applied (CR), one to which the communication reduction and vertex inactivation were applied (CR+VI), and a handwritten Giraph program.

We prepared three graphs: WebBerkStan, RoadNet-PA, and Rand1M-10M (Table 1). The first two were obtained from the Stanford Large Network Dataset Collection⁴. The former is a web graph; the latter is a road network. The last one

³ Z3 Solver: <https://z3.codeplex.com/>.

⁴ <https://snap.stanford.edu/data/>.

Table 2. Performances of programs (unit: seconds)

	Original	CR	CR+VI	Handwritten
SSSP/WebBerkStan	233.2	54.2	46.0	26.8
SSSP/RoadNet-PA	146.4	70.8	47.2	32.1
SSSP/Rand-1M10M	16.9	7.3	7.4	4.6
PageRank/WebBerkStan	20.8	–	–	12.7
PageRank/RoadNet-PA	12.6	–	–	7.6
PageRank/Rand-1M10M	26.1	–	–	17.1
SCC/WebBerkStan	1765.2	1413.1	–	254.9
SCC/RoadNet-PA	326.7	154.9	–	55.5
SCC/Rand-1M10M	35.4	28.1	–	12.6

is a graph generated by randomly connecting vertices. All graphs are directed. RoadNet-PA is symmetric, i.e., each edge accompanies the reverse edge.

The environment for the experiment was a PC cluster consisting of 16 computational nodes. Each node consisted of Intel Core i5 CPUs (nine of them were Core i5-2500, and the other seven were i5-7600), 8-GB memory, and a 128-GB SSD. As the backend of Fregel, we used Giraph 1.3.0, Hadoop 1.2.1, and Java 1.8.0_141 running on Debian 4.9.6-3. We used 16 workers for each experiment.

4.3 Results

For all programs, optimizations were performed immediately (within 0.1 s). For SSSP, both communication reduction and vertex inactivation were possible. For PageRank, both optimizations were impossible. For SCC, although the optimizer guaranteed that both optimizations were possible, the Fregel compiler could not introduce vertex inactivation because Giraph does not support vertex reactivation after all the vertices become inactive. If vertices are inactivated based on Property 3, after finding a strongly connected component, they should be reactivated to find another strongly connected component. The handwritten Giraph program instead inactivates vertices that are removed from the graph. This optimization is impossible based on Property 3 because its justification requires an analysis beyond a single `fregel` function.

As shown in Table 2, the original program for SSSP were significantly slower than the handwritten program. Our optimization removed most of the inefficiencies, leading to a program that were roughly only 1.5 times as slow as the handwritten one. Although our method was not able to optimize PageRank, the difference between the original and the handwritten programs was relatively small. For SCC, while the communication reduction was effective, absence of the vertex inactivation make the optimized program less efficient than the case of SSSP. The program is especially slow for WebBerkStan. We guess that the inefficiency comes from the iterative nature of the SCC algorithm, which requires

a lot of supersteps (thereby synchronization barriers) for analyzing graphs that contain many strongly connected components.

Possibility of Other Optimizations. For SSSP and SCC, the other optimizations, barrier removal and prioritized execution, are theoretically applicable. They may be effective especially for SCC. The maximum vertex id is intensively propagated without being interrupted by barriers. Lemma 2 cannot be applied to PageRank. Existing asynchronous implementations of PageRank use the previous messages of neighbors if new ones have not yet arrived. Lemma 2 considers processing computations using arrived messages only.

5 Related Work

As mentioned in the introduction, the optimizations discussed are not new. Vertex inactivation is a part of the core functionality of Pregel [1]. The communication reduction technique for SSSP was also reported [1]. Many VcGP frameworks use asynchronous execution [15–17]; moreover, some combine asynchronous and synchronous execution to further improve efficiency [2,3]. Some frameworks [3,6,7] support prioritized execution as well. The effectiveness of these optimizations has been intensively studied. Our contribution is their automation using constraint solvers.

Modern constraint solving techniques including QE and SMT have been used for program analysis and synthesis [18]. A typical application is optimization of nested loops, especially *stencil* loops [19–21]. Our optimization can be understood as a variant of such loop optimizations. For instance, introduction of asynchronous execution is essentially an exchange of the inner and outer loops, which is a typical application of constraint solving techniques. From this perspective, the distinctive feature of our study is that it deals with graph manipulation programs. Graph manipulation tends to form irregular complex loops and may not be captured by formalisms supported by constraint solvers, e.g., a system of linear inequalities. Our study focused on VcGP rather than general graph processing and provided a supporting lemmas (Lemmas 1 and 2) that enable constraint solvers to perform optimizations.

Most related systems are Elixir [6,22] and Distributed Socialite [23]. Elixir automatically derives efficient distributed graph processing from the logical specifications of the output graph. It uses an SMT solver to specify the vertices that should be processed at each step. Distributed Socialite is a graph processing language similar to Datalog. It accelerates SSSP-like computation by using the generalized Δ -stepping algorithm [24], in which vertices are processed according to a special priority, if a certain kind of monotonicity property is detected. Both start from declarative programs and apply nontrivial optimizations by analyzing certain properties. Unfortunately, both require programmers to provide some clues for optimizations. For instance, with Elixir, programmers should specify the conditions for sending messages and the priorities for processing vertices. With Distributed Socialite, the generalized Δ -stepping is applied only if programmers

use certain operators. In addition, both frameworks are based on asynchronous execution. We have shown that intensive use of constraint solvers enables many interesting optimizations to be applied to nearly annotation-free deterministic programs.

6 Conclusion and Future Work

We have developed a method of automatically applying nontrivial optimizations to declarative VcGP programs. The key is the use of constraint solvers to reveal the program properties. In our experiments, optimizations were achieved within a realistic time frame and led to significant performance improvement.

We are developing another backend of Fregel based on Pregel+⁵. The new backend will enable more rigorous evaluation of our method.

Our approach to optimize Fregel programs can be used for other declarative graph processing frameworks [6, 7, 22, 23, 25]. These frameworks generally require users to write specific programs (e.g., adding annotations and/or invoking certain API functions) in order to apply nontrivial optimizations. It would be interesting if constraint solvers enabled these optimizations to be applied to naively written programs.

Acknowledgements. The authors are grateful to Shigeyuki Sato for discussion with him about related work. This study is partly supported by JSPS Kakenhi JP26280020 and JP15K15965.

References

1. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Elmagarmid, A.K., Agrawal, D. (eds.) *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010*, pp. 135–146. ACM (2010)
2. Xie, C., Chen, R., Guan, H., Zang, B., Chen, H.: SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In: Cohen, A., Grove, D. (eds.) *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2015*, pp. 194–204. ACM (2015)
3. Liu, Y., Zhou, C., Gao, J., Fan, Z.: Giraphasync: supporting online and offline graph processing via adaptive asynchronous message processing. In: Mukhopadhyay, S., Zhai, C., Bertino, E., Crestani, F., Mostafa, J., Tang, J., Si, L., Zhou, X., Chang, Y., Li, Y., Sondhi, P. (eds.) *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016*, pp. 479–488. ACM (2016)
4. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From “think like a vertex” to “think like a graph”. *PVLDB* **7**(3), 193–204 (2013)
5. Quamar, A., Deshpande, A., Lin, J.J.: NScale: neighborhood-centric large-scale graph analytics in the cloud. *VLDB J.* **25**(2), 125–150 (2016)


⁵ Pregel+: www.cse.cuhk.edu.hk/pregelplus/.

6. Proutzoz, D., Manevich, R., Pingali, K.: Elixir: a system for synthesizing concurrent graph programs. In: Leavens, G.T., Dwyer, M.B. (eds.) Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, pp. 375–394. ACM (2012)
7. Cruz, F., Rocha, R., Goldstein, S.C.: Declarative coordination of graph-based parallel programs. In: Asenjo, R., Harris, T. (eds.) Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2016, pp. 4:1–4:12. ACM (2016)
8. Emoto, K., Matsuzaki, K., Hu, Z., Morihata, A., Iwasaki, H.: Think like a vertex, behave like a function! A functional DSL for vertex-centric big graph processing. In: Garrigue, J., Keller, G., Sumii, E. (eds.) Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, pp. 200–213. ACM (2016)
9. Caviness, B.F., Johnson, J.R. (eds.): Quantifier Elimination and Cylindrical Algebraic Decomposition. Springer, Vienna (1998). <https://doi.org/10.1007/978-3-7091-9459-1>
10. de Moura, L.M., Bjørner, N.: Satisfiability modulo theories: introduction and applications. *Commun. ACM* **54**(9), 69–77 (2011)
11. Valiant, L.G.: A bridging model for parallel computation. *Commun. ACM* **33**(8), 103–111 (1990)
12. Yan, D., Cheng, J., Lu, Y., Ng, W.: Effective techniques for message reduction and load balancing in distributed graph computation. In: Gangemi, A., Leonardi, S., Panconesi, A. (eds.) Proceedings of the 24th International Conference on World Wide Web, WWW 2015, pp. 1307–1317. ACM (2015)
13. Verma, S., Leslie, L.M., Shin, Y., Gupta, I.: An experimental comparison of partitioning strategies in distributed graph processing. *PVLDB* **10**(5), 493–504 (2017)
14. Salihoglu, S., Widom, J.: Optimizing graph algorithms on pregel-like systems. *PVLDB* **7**(7), 577–588 (2014)
15. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: distributed graph-parallel computation on natural graphs. In: Thekkath, C., Vahdat, A. (eds.) Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, USENIX Association, pp. 17–30 (2012)
16. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning in the cloud. *PVLDB* **5**(8), 716–727 (2012)
17. Han, M., Daudjee, K.: Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB* **8**(9), 950–961 (2015)
18. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. *Found. Trends Program. Lang.* **4**(1–2), 1–119 (2017)
19. Gröbflinger, A., Griebel, M., Lengauer, C.: Quantifier elimination in automatic loop parallelization. *J. Symb. Comput.* **41**(11), 1206–1221 (2006)
20. Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 132–146. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78791-4_9

21. Pouchet, L., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P., Vasilache, N.: Loop transformations: convexity, pruning and optimization. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 549–562. ACM (2011)
22. Proutzoz, D., Manevich, R., Pingali, K.: Synthesizing parallel graph programs via automated planning. In: Grove, D., Blackburn, S. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 533–544. ACM (2015)
23. Seo, J., Park, J., Shin, J., Lam, M.S.: Distributed socialite: a datalog-based language for large-scale graph analysis. PVLDB **6**(14), 1906–1917 (2013)
24. Meyer, U., Sanders, P.: [Delta]-stepping: a parallelizable shortest path algorithm. J. Algorithms **49**(1), 114–152 (2003)
25. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: GraphX: graph processing in a distributed dataflow framework. In: Flinn, J., Levy, H. (eds.) Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014, pp. 599–613. USENIX Association (2014)



Breaking Symmetries with Lex Implications

Michael Codish¹, Thorsten Ehlers²(✉), Graeme Gange³, Avraham Itzhakov¹,
and Peter J. Stuckey^{3,4} 

¹ Department of Computer Science, Ben-Gurion University of the Negev,
Beersheba, Israel

² Department of Computer Science, Kiel University, Kiel, Germany
`the@informatik.uni-kiel.de`

³ Department of Computing and Information Systems,
The University of Melbourne, Melbourne, Australia

⁴ Data61 CSIRO, Melbourne, Australia

Abstract. Breaking symmetries is crucial when solving hard combinatorial problems. A common way to eliminate symmetries in CP/SAT is to add symmetry breaking constraints. Ideally, symmetry breaking constraints should be complete and compact. The aim of this paper is to find compact and complete symmetry breaks applicable when solving hard combinatorial problems using CP/SAT approach. In particular: graph search problems and matrix model problems where symmetry breaks are often specified in terms of lex constraints. We show that sets of lex constraints can be expressed with only a small portion of their inner lex implications which are a particular form of Horn clauses. We exploit this fact and compute a compact encoding of the row-wise LexLeader and state of the art partial symmetry breaking constraints. We illustrate the approach for graph search problems and matrix model problems.

1 Introduction

When solving hard combinatorial problems, symmetry breaks play a crucial role. When seeking solutions, the size of the search space is significantly reduced if symmetries are eliminated. The search space can be explored more efficiently when avoiding paths that lead to symmetric solutions and avoiding also those that lead to symmetric non-solutions.

This paper deals with *variable symmetry* in constraint satisfaction problems (CSP) where symmetry is a permutation defined over a set of variables that preserves solutions. Given a CSP with variables x_1, \dots, x_n , we say that σ is a symmetry if for every assignment $\mu = \{x_1 = i_1, \dots, x_n = i_n\}$, μ is a solution if and only if $\{x_1 = i_{\sigma(1)}, \dots, x_n = i_{\sigma(n)}\}$ is also a solution.

Supported by the Israel Science Foundation, grant 625/17 and the German Federal Ministry of Education and Research, combined project 01IH15006A.

One common approach to eliminate symmetries is to introduce symmetry breaking constraints [1–4] which rule out isomorphic solutions thus reducing the size of the search space while preserving the set of solutions. Ideally, a symmetry breaking constraint is satisfied by a single member of each equivalence class of solutions, in which case it is said to be *complete*. However, computing such symmetry breaking constraints is, most often, intractable [2]. In practice, symmetry breaking constraints are often *partial*, and typically rule out some, but not all of the symmetries in the search. As noted in the survey by Walsh [5], often a few simple constraints rule out most of the symmetries.

In many cases, symmetry breaking constraints, complete or partial, are expressed in terms of lex constraints on the variables of the problem. Each lex constraint, corresponds to one symmetry σ , and restricts the search space to consider assignments which are lexicographically smaller than their permuted form obtained according to σ . Typical examples are: graph search problems [6] where rows and columns of the Boolean adjacency matrix can be reordered by some permutation, and matrix models where rows and columns can be reordered by a pair of permutations. For matrix models common partial symmetry breaks described in terms of lex constraints include doubleLex [7] (denoted also lex^2 in [8]) and snake-lex [9]. For graph search problems, Codish *et al.* [6] introduce partial symmetry breaks (denoted sb_ℓ and sb_ℓ^*) which are refinements of doubleLex for adjacency matrices.

Complete symmetry breaks can be obtained, for both types of problems, by introducing a lex constraint for each reordering of the combinatorial object (graph or matrix) [10]. For matrices, the reordering takes place by permuting rows and columns [7]. For graphs, symmetric solutions can be obtained by permutations of the vertices, which corresponds to simultaneously permuting both rows and columns of the adjacency matrix. However, the number of lex constraints is overwhelming.

The aim of this research is to find compact and complete symmetry breaks applicable when solving hard combinatorial problems. In particular: graph search problems and matrix model problems.

In previous work, Itzhakov and Codish [11] present complete and compact symmetry breaks for graphs based on so-called canonizing sets of permutations where each permutation represents a lex constraint. Their approach is based on the observation that many of the lex constraints expressed in terms of all permutations (of rows and columns of the adjacency matrix) are redundant. Itzhakov and Codish [11] compute compact symmetry breaking constraints for graphs with 10 or less vertices. They observe, for example, that for 10 vertices 7,853 lex constraints suffice to provide a complete symmetry break instead of the $10! = 3,628,800$ constraints introduced by the definition. Itzhakov and Codish [11] report that this symmetry break takes 4 days to compute.

Heule [12] poses the question: How expensive is it to break all graph symmetries? Heule seeks an answer in terms of the number of clauses in a CNF representation of the corresponding symmetry breaking constraint. For up to $n = 5$ vertices, Heule computes size-optimal compact and complete symmetry

breaks. A size-optimal complete symmetry break for graphs with 5 vertices consists of only 12 clauses. In contrast, the symmetry break computed by Itzhakov and Codish consists of 7 lex constraints, which can be encoded in 83 clauses. For $5 < n \leq 8$ vertices, Heule computes complete symmetry breaks which are significantly smaller than those computed by Itzhakov and Codish but which are not determined to be optimal. For 8 vertices, the complete symmetry break computed by Heule consists of 956 clauses (and takes two days to compute). The complete symmetry break computed by Itzhakov and Codish consists of 135 lex constraints (2724 clauses) and as reported in [11] takes 6 min to compute.

Frisch and Harvey illustrate in [13] the redundancies in a complete symmetry break in a three-by-two matrix. They show how to simplify the 11 lex constraints expressing all reorderings of rows and columns. The resulting symmetry break has 8 simplified lex constraints.

In this paper we note the standard decomposition of a lex constraint of the form $x_1 \dots x_n \leq_{lex} y_1 \dots y_n$ to a conjunction of Horn clauses of the form

$$(x_1 = y_1), \dots, (x_k = y_k) \rightarrow x_{k+1} \leq y_{k+1}$$

where the literals on the left side are equalities between the variables in the lex constraint and $1 \leq k < n$ [14]. We call clauses of this form *lex implications*. We observe that many of the lex implications in the decomposition of the complete symmetry breaks derived in [11] are redundant. This enables to significantly reduce the size of the symmetry breaking constraints. For example, for $n = 10$ vertices, a complete symmetry break is obtained in [11] with 7,853 lex constraints. These decompose to 248,604 lex implications which can be reduced (removing implications implied by the others) to a complete symmetry break expressed using only 21,844 lex implications. For 5 vertices, the complete symmetry break computed in [11] involves 7 lex constraints which decompose to 41 lex implications. These can be reduced to 14 non-redundant lex implications and 40 clauses, cf. Example 3. For 8 vertices, the complete symmetry break computed in [11] involves 135 lex constraints which decompose to 2006 lex implications (2724 clauses). These can be reduced to 387 non-redundant lex implications (1077 clauses), c.f. Table 2.

Given the observation that so many lex implications are redundant, we then pose the direct question: how many lex implications are required to express a complete symmetry break on a graph with n nodes. We generate such symmetry breaks directly and not by reducing the complete symmetry breaks presented in [11]. A compact and complete symmetry break for graphs with 8 vertices can be computed in about 1 min. A compact and complete symmetry break for 10 vertices can now be computed in roughly 3 h (in contrast to 4 days as reported in [11]). Moreover, we compute a compact and complete symmetry break for graphs with 11 vertices. This symmetry break consists in 274,109 lex implications (280,049 clauses) and is computed in 8 days.

The technique of representing conjunctions of lex constraints in terms of non-redundant lex implications works also for matrix models where symmetry breaks are also defined in terms of lex constraints (swapping rows and columns) and also for partial symmetry breaks for graphs and matrix models.

We analyze the standard doubleLex constraint and observe that it has no redundancies when represented as lex implications. However when using extensions which we call lex^+ and lex^* (also known as swapNext and swapAny respectively [15]) there are many redundancies. We give compact versions of these (without the redundancies).

The rest of the paper is organized as follows. In the next section we introduce notation and basic concepts. In Sect. 3 we illustrate the effect of removing redundancy in encoding symmetry breaking constraints. In Sect. 4 we define our approach to efficiently generating compact and complete symmetry breaking constraints, and illustrate the effectiveness on graphs. Finally in Sect. 6 we conclude.

The computations throughout the paper are performed using the finite-domain constraint compiler BEE [16] which compiles constraints to CNF, and solves it applying an underlying SAT solver. We use Glucose 4.0 [17] as the underlying SAT solver except where specified that we used Clasp 3.1.3 [18]. All computations were performed on an Intel E8400 core, clocked at 2 GHz, able to run a total of 12 parallel threads. Each of the cores in the cluster has computational power comparable to a core on a standard desktop computer. Each SAT instance is run on a single thread, and all running times reported in this paper are CPU times.

2 Preliminaries

In this paper, we consider Boolean formulas φ which encode the set of solutions of combinatorial problems. In many cases, there are variable symmetries in the solution space of such problems. This is, given a solution $x_i = v_i$ there often exist permutations π such that $x_{\pi(i)} = v_i$ is an isomorphic solution [4].

When enumerating the solutions for a particular problem, it is often preferable to consider only non-isomorphic solutions. Furthermore, if one wishes to prove that no solutions for some problem exist, breaking symmetries often allows for smaller proofs.

In order to decrease the size of the solution space, one approach is to use constraints ψ which break symmetries in the solution space of φ . This is, for every solution x which satisfies φ , there exists an isomorphic solution x' which satisfies $\varphi \wedge \psi$. Symmetry breaking constraints which rule out all but one solution from each equivalence class are called complete. Symmetry breaking constraints which rule out some but not all isomorphic solutions from an equivalence class are called partial [2].

We consider finite and simple graphs (undirected with no self loops). The set of simple graphs on n nodes is denoted \mathcal{G}_n . We assume that the vertex set of a graph, $G = (V, E)$, is $V = \{1, \dots, n\}$ and represent G by its $n \times n$ Boolean adjacency matrix. We often let G denote both the graph itself and also its adjacency matrix.

Graph search problems are about the search for a graph which satisfies a given set of constraints, φ , or to determine that no such graph exists. In this

$$A = \begin{bmatrix} 0 & a & b & c & d \\ a & 0 & e & f & g \\ b & e & 0 & h & i \\ c & f & h & 0 & j \\ d & g & i & j & 0 \end{bmatrix} \qquad \pi(A) = \begin{bmatrix} 0 & a & e & f & g \\ a & 0 & b & c & d \\ e & b & 0 & h & i \\ f & c & h & 0 & j \\ g & d & i & j & 0 \end{bmatrix}$$

Fig. 1. The 5×5 Boolean adjacency matrix G and $\pi(G)$ for $\pi = (2, 1, 3, 4, 5)$.

setting the unknown graph is represented by an adjacency matrix consisting of Boolean variables which are constrained by φ . Often graph search problems are about the search for the set of all graphs, modulo graph isomorphism, that satisfy the given constraints.

The set of permutations $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ is denoted S_n . Permutations act on adjacency matrices in the natural way: If A is the adjacency matrix of a graph G and π is a permutation, then $\pi(A)$ is the adjacency matrix obtained by simultaneously permuting with π the rows and columns of A .

Two graphs $G_1, G_2 \in \mathcal{G}_n$ are isomorphic, denoted $G_1 \approx G_2$, if there exists a permutation $\pi \in S_n$ such that $G_1 = \pi(G_2)$. Sometimes we write $G_1 \approx_\pi G_2$ to emphasize that π is the permutation such that $G_1 = \pi(G_2)$. For sets of graphs H_1, H_2 , we say that $H_1 \approx H_2$ if for every $G_1 \in H_1$ (likewise in H_2) there exists $G_2 \in H_2$ (likewise in H_1) such that $G_1 \approx G_2$.

We consider an ordering on graphs, defined by representing their adjacency matrices as strings. Because adjacency matrices are symmetric with zeroes on the diagonal, it suffices to focus on the upper triangle parts of the matrices [19].

Definition 1 (ordering graphs). Let $G_1, G_2 \in \mathcal{G}_n$ and let s_1, s_2 be the strings obtained by concatenating the rows of the upper triangular parts of their corresponding adjacency matrices A_1, A_2 respectively. Then, $G_1 \preceq G_2$ if and only if $s_1 \preceq_{lex} s_2$. We also write $A_1 \preceq A_2$.

A classic complete symmetry break for graphs is the LEXLEADER constraint [10] defined as follows:

Definition 2 (LexLeader). $\text{LEXLEADER}(n) = \bigwedge \{ G \preceq \pi(G) \mid \pi \in S_n \}$ where G is an $n \times n$ matrix of Boolean variables with 0's on the diagonal and such that $G_{ij} = G_{ji}$ for all $1 \leq i < j \leq n$. Sometimes we write $\text{LEXLEADER}_G(n)$ to make G explicit.

Example 1. Consider graphs on 5 nodes. Figure 1 depicts the adjacency matrix A of such graphs, where $a \dots j$ denote Boolean variables (on the left side). For the permutation $\pi = (2, 1, 3, 4, 5)$, $\pi(A)$, is detailed on the right side of Fig. 1. A complete symmetry break can be created by using the LEXLEADER constraint, which requires $5! = 120$ lex constraints, one for each permutation in S_5 .

The constraint $G \preceq \pi(G)$ is expressed as the lex constraint $abcdefghij \preceq_{lex} aefgbc dhij$, which can be simplified to $bcd \preceq_{lex} efg$.

$\pi_1 = (5, 3, 4, 2, 1)$	$abcefgi \preceq_{lex} ijghebc$
$\pi_2 = (2, 1, 5, 3, 4)$	$bcdefhi \preceq_{lex} gefdbij$
$\pi_3 = (1, 3, 2, 4, 5)$	$afg \preceq_{lex} bhi$
$\pi_4 = (1, 2, 4, 3, 5)$	$bei \preceq_{lex} cfj$
$\pi_5 = (2, 4, 1, 5, 3)$	$abcdefgi \preceq_{lex} fagecjh$
$\pi_6 = (2, 3, 1, 5, 4)$	$abcdfgh \preceq_{lex} eagfihd$
$\pi_7 = (1, 2, 5, 3, 4)$	$bcefhi \preceq_{lex} dbgeij$

Fig. 2. A complete symmetry break for graphs with 5 nodes expressed in terms of 7 lex constraints derived from corresponding permutations.

$a \leq b$	$(T_{a=h}) \Rightarrow c \leq i$	$(T_{b=c} \wedge T_{e=f}) \Rightarrow i \leq j$
$b \leq c$	$(T_{b=f}) \Rightarrow c \leq e$	$(T_{a=b} \wedge T_{f=h}) \Rightarrow g \leq i$
$c \leq d$	$(T_{c=d}) \Rightarrow f \leq g$	$(T_{b=g} \wedge T_{c=e}) \Rightarrow d \leq f$
$b \leq f$	$(T_{a=b}) \Rightarrow f \leq h$	$(T_{b=f} \wedge T_{c=e} \wedge T_{d=g}) \Rightarrow i \leq j$
	$(T_{b=c}) \Rightarrow e \leq f$	$(T_{b=e} \wedge T_{c=g} \wedge T_{d=f}) \Rightarrow h \leq i$

Fig. 3. A complete symmetry break for graphs with 5 nodes expressed in terms of 14 lex implications. These clauses were computed directly, they are not derived from the lex constraints presented in Fig. 2.

In fact, it is sufficient to consider only some of the LEXLEADER constraints. In [11], a refinement procedure was used which adds non-redundant lex constraints until a complete symmetry break has been reached.

Example 2. A complete symmetry break for graph search problems on 5 nodes can be expressed in terms of the 7 permutations detailed in Fig. 2 (on the left) which give rise to the corresponding lex constraints (on the right). All of the $5! = 120$ lex constraints used by the LEXLEADER(5) constraint are implied by these 7 lex constraints.

It is well known that lex constraints can be decomposed into lex implications. Using Tseytin variables $T_{x=y} \leftrightarrow x = y$ and replacing $a \leq b$ by $(\neg a \vee b)$, these are Horn clauses. The question we ask in this paper is: How many lex implications are required to represent a complete symmetry break on graphs?

Example 3. In order to break all symmetries on graphs with 5 nodes, it is sufficient to consider the 14 lex implications depicted in Fig. 3. As the Tseytin variables only occur on the left-hand side of the implications, it is sufficient to encode them using two clauses as in $(x \wedge y) \Rightarrow T_{x=y}$ and $(\neg x \wedge \neg y) \Rightarrow T_{x=y}$ [20]. Thus, this symmetry break can be encoded using 40 clauses.

In some cases, the redundancy of lex implications can be seen directly. The lex constraint $abcefgi \preceq_{lex} ijghebc$ from Example 2 implies $a \leq i$, which is redundant with respect to the lex implications from Fig. 3: If $a = 1$, this implies $b = c = d = f = 1$ by the inequalities on the left-hand side. This again implies

$h = 1$ by the lex implication $(T_{a=b}) \Rightarrow f \leq h$, and $i = 1$ by $(T_{a=h}) \Rightarrow c \leq i$. Checking the redundancy of other lex implications often requires a case analysis.

In this paper we consider several partial symmetry breaks for graphs and for matrix models which can be defined in terms of specific sets of permutations. We denote by S_n^{adj} and by S_n^{pair} the sets of permutations on $\{1, \dots, n\}$ which swap a single adjacent pair $(i, i + 1)$ for $1 \leq i < n$ and respectively a single pair (i, j) for $1 \leq i < j \leq n$.

The partial symmetry breaks sb_ℓ and sb_ℓ^* presented in [6] for graphs are defined as follows where A is an $n \times n$ adjacency matrix of Boolean variables:

$$sb_\ell(A) = \bigwedge_{\pi \in S_n^{adj}} (A \leq \pi(A)) \quad \text{and} \quad sb_\ell^*(A) = \bigwedge_{\pi \in S_n^{pair}} (A \leq \pi(A))$$

Thus, they can be encoded using $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$ lex constraints, respectively.

Definition 3 (ordering matrices). *Let M_1, M_2 be a pair of $m \times n$ matrices of Boolean variables and let s_1, s_2 be the strings obtained by concatenating their rows respectively. Then, $M_1 \preceq M_2$ if and only if $s_1 \preceq_{lex} s_2$.*

For a $m \times n$ matrix M of Boolean variables and permutations $\pi_1 \in S_m$, $\pi_2 \in S_n$, let $\pi_1^{rows}(M)$ denote the matrix obtained by permuting the rows of M by π_1 and let $\pi_2^{cols}(M)$ denote the matrix obtained by permuting the columns of M by π_2 . The doubleLex symmetry break, also denoted lex^2 [7] is defined by

$$lex^2(M) = \bigwedge \left(\left\{ (M \leq \pi^{rows}(M)) \mid \pi \in S_m^{adj} \right\} \cup \left\{ (M \leq \pi^{cols}(M)) \mid \pi \in S_n^{adj} \right\} \right)$$

It enforces rows and columns to be sorted lexicographically and it can be encoded using $\mathcal{O}(n + m)$ lex constraints. We also consider extensions of lex^2 , denoted lex^+ and lex^* .

$$lex^+(M) = lex^2(M) \wedge \bigwedge \left\{ (M \leq \pi_1^{rows} \pi_2^{cols}(M)) \mid \pi_1 \in S_m^{adj}, \pi_2 \in S_n^{adj} \right\}$$

$$lex^*(M) = lex^2(M) \wedge \bigwedge \left\{ (M \leq \pi_1^{rows} \pi_2^{cols}(M)) \mid \pi_1 \in S_m^{pair}, \pi_2 \in S_n^{pair} \right\}$$

Encoding these symmetry breaks requires $\mathcal{O}(nm)$ and $\mathcal{O}(n^2m^2)$ lex constraints, respectively.

In [8], the authors suggest combining lex^2 with additional constraints which enforce that the first row is lexicographically smaller than every permutation of every other row, and call this symmetry break allPerm. It can be implemented to run in linear time, however, encoding it statically into a SAT formula requires $\mathcal{O}(n!m)$ lex constraints. As we will show in Sect. 3, most of these constraints are actually redundant.

Table 1 illustrates the relative power of several symmetry breaks for Boolean matrix models. We consider matrices of size $n \times n$ and report the number of solutions for each of the symmetry breaks. The smaller the solution, the more precise the symmetry break. The symmetry breaks are detailed from weakest

Table 1. The number of solutions for Boolean matrix models with various symmetry breaks.

n	None	lex ²	allPerm	lex ⁺	lex [*]	Complete
3	2 ⁹	45	41	37	36	36
4	2 ¹⁶	650	520	366	330	317
5	2 ²⁵	24,520	17,128	8,659	6,779	5,624
6	2 ³⁶	2,625,117	1,616,074	602,813	391,532	251,610
7	2 ⁴⁹	836,488,618	458,375,316	139,268,908	73,720,859	33,642,660

(left) to strongest (right). The left column, titled “None” has the most solutions and corresponds to imposing no symmetry break. The right column, titled “Complete” has the least solutions and corresponds to imposing a complete symmetry break. This column is obtained as OEIS sequence A002724 [21]. We observe that allPerm is only slightly stronger than lex² and weaker than lex⁺. This is surprising, as lex⁺ is polynomial in size whilst allPerm is exponential.

3 Removing Redundant Constraints

In [11], the authors generated complete symmetry breaks for graph problems. They aimed for a small set of permutations which is canonizing, i.e. lex constraints derived from them create a complete symmetry break. They found that while generating such sets, some of the lex constraints became redundant and could be removed. Thus, after generating a complete symmetry break, they removed as many lex constraints as possible, and derived a set of non-redundant lex constraints. They furthermore noted that the set of permutations required for a complete symmetry break for graph problems on n nodes has significantly less than $n!$ elements.

Here, we consider the set of lex implications derived from a set of lex constraints rather than the lex constraints themselves. We show that even if the set of lex constraints is non-reducible, many of the lex implications are redundant and can be removed. The approach of removing redundant lex implications applies both to complete and partial symmetry breaks.

Table 2 shows the size of different symmetry breaks for graphs, both in terms of lex implications, and in terms of the number of clauses (in parentheses). This includes clauses required for encoding the Tseytin variables. For each symmetry break, the table details the impact of removing redundant lex implications. The columns titled “orig” denote the size of the symmetry breaks before reduction (obtained by decomposing the lex constraints), and the columns titled “red” denote the size of the symmetry breaks after removing redundant lex implications.

The constraints sb_ℓ and sb_ℓ^* are partial symmetry breaks introduced in [6]. Interestingly, sb_ℓ does not contain any redundant lex implications, whereas roughly 65% of the lex implications of sb_ℓ^* are redundant. For the right-most

Table 2. Number of lex implications and clauses (in parentheses), before and after the reduction, for partial symmetry breaks sb_ℓ , sb_ℓ^* (on graphs) and for a complete symmetry break (for graphs) based on canonizing sets.

n	sb_ℓ	sb_ℓ^*		Canonizing	
	orig & red	orig	red	orig	red
3	2(2)	3(3)	2(2)	3(2)	3(2)
4	6(12)	12(24)	6(12)	6(12)	6(12)
5	12(28)	30(70)	13(33)	41(83)	21(55)
6	20(50)	60(150)	24(72)	70(156)	38(118)
7	30(78)	105(273)	40(136)	302(580)	108(374)
8	42(112)	168(448)	62(232)	2006(2724)	387(1077)
9	56(152)	252(684)	91(367)	17059(18311)	2366(3600)
10	72(198)	360(990)	128(548)	248604(250582)	21844(23814)

Table 3. Number of lex implications for different symmetry breaks for matrix models before and after reduction.

n	lex^2		lex^+		lex^*		allPerm	
	orig	reduced	orig	reduced	orig	reduced	orig	reduced
3	12	12	28	16	64	16	48	13
4	24	24	78	37	294	45	312	32
5	40	40	168	77	968	112	2440	71
6	60	60	310	141	2560	252	21660	148
7	84	84	516	235	5808	532	211764	310
8	112	112	798	365	11774	1048	—	—
9	144	144	1168	536	21904	1944	—	—
10	180	180	1638	755	38088	3413	—	—

column, we took canonizing sets from [11], translated them into lex implications, and removed redundant clauses. Although the set of lex constraints does not contain any redundant constraints, more than 90% of the lex implications could be removed for $n = 10$ nodes. Furthermore, it is noteworthy that on small graphs, there are more clauses for the Tseytin encoding than for the symmetry break. For larger graphs, most of the clauses are lex implications.

Table 3 illustrates the reduction in size of symmetry breaks for Boolean matrix models of size $n \times n$. Here we focus on the number of lex implications in the symmetry break (before and after reduce). In the table we consider the symmetry breaks lex^2 , lex^+ , lex^* and allPerm which are described in Sect. 2.

We observe that DoubleLex (lex^2) does not contain any redundant implications. On the contrary, more than half of the lex implications from lex^+ are redundant and approximately 90% of the lex implications in lex^* are redundant.

With regards to the allPerm symmetry break proposed in [8], the constraint itself is huge. We did not generate it for matrices larger than 7×7 for which 99.85% of the lex implications are redundant. This huge size makes it hard to compute a reduced set of lex implications, we refrained from investing computational resources for the reduction of allPerm on matrices of size 8×8 and larger.

How the Reduce Works

Basically, our algorithm iterates over the set of lex implications, and checks for each of them if they are redundant. This is done by removing them from the formula, and checking if there is a solution which would be forbidden by this clause, as shown in Algorithm 2. If this is not the case, the clause is redundant and can be removed. Contrary to other approaches like the one presented in [22], we run a full SAT search to determine if a lex implication is actually redundant or not. This allows for removing more clauses. Furthermore, the number of clauses which can actually be removed depends on the order in which clauses are checked. Some clauses are more helpful as they contribute to making other clauses redundant. Thus, we run our reduction in two phases. The first phase is shown in Algorithm 1. Here, we check if a clause c is redundant. If this is the case, we compute a subset $\psi \subseteq \varphi'$ of clauses which makes c redundant, and increase the ranking of all clauses within this set. The rationale is that removing these clauses is more likely make other clauses no longer redundant, and so increase the size of the final symmetry break.

In the second stage, we sort the clauses by ranking, so clauses which were frequently the cause of redundancy appear as late as possible. We then reduce the set of lex implications using Algorithm 2.

Algorithm 1. Ranking Redundant Constraints

```

rank( $c$ ) = 0  $\forall c \in \varphi$ 
for all  $c \in \varphi$  do
   $\varphi' = (\varphi \setminus \{c\}) \cup \neg c$ 
  if UNSAT( $\varphi'$ ) then
    Let  $\psi \subseteq \varphi'$  such that  $\psi \wedge \neg c \equiv \perp$ 
    for all  $c' \in \psi$  do
      rank( $c'$ )  $\leftarrow$  rank( $c'$ ) + 1

```

Sort clauses by their ranks, small ranks first.

Algorithm 2. Removing Redundant Constraints

```

Rank clauses with Algorithm 1
for all  $c \in \varphi$  in ascending order do
   $\varphi' = (\varphi \setminus \{c\}) \cup \neg c$ 
  if UNSAT( $\varphi'$ ) then
     $\varphi = \varphi \setminus \{c\}$ 

```

4 Generating Compact and Complete Symmetry Breaks for Graphs

The LexLeader constraint which is a complete symmetry break defined in terms of all permutations of a graph can be expressed as a set of lex implications. Each lex constraint $G \leq_{lex} \pi(G)$, for a permutation π is decomposed to lex implications, as described in Sect. 2. Each implication is classified by two parameters: the length of the implication and the permutation from which the originating lex constraint was generated. The length of an implication is the number of atoms it contains which is between 1 and $\binom{n}{2}$ (the size of the upper triangle of the adjacency matrix). Formally, let A be an $n \times n$ matrix, $\pi \in S_n$ a permutation, and $1 \leq k \leq \binom{n}{2}$ an implication length. Let $x_1, \dots, x_{\binom{n}{2}}$ and $y_1, \dots, y_{\binom{n}{2}}$ be the upper triangle elements (row by row) of A and $\pi(A)$, respectively. The length k lex implication $Imp_A(k, \pi)$ is defined by

$$Imp_A(k, \pi) = (x_1 = y_1) \wedge \dots \wedge (x_{k-1} = y_{k-1}) \Rightarrow x_k \leq y_k$$

Using this notation the classic LexLeader constraint for an $n \times n$ adjacency matrix A is equivalent to the following lex implication representation:

$$LexLeader_A(n) = \bigwedge_{\pi \in S_n} \bigwedge_{k=1}^{\binom{n}{2}} Imp_A(k, \pi)$$

In this work we generate a complete symmetry breaking constraint that is equivalent to LexLeader by repeatedly selecting lex implications from the definition of $LexLeader_A(n)$ which are not logically implied by those already selected. When no further lex implications can be selected we have found a complete symmetry break. Although we repeatedly select non-redundant lex implications, it is possible, because of the order of selection, that some of the implications in the set become redundant. For this we reason we perform a second pass to repeatedly remove redundant implications. This process is formalized as Algorithm 3 where we select implications according to their length, first the short ones, and then the longer ones. To derive a complete symmetry break for graphs with n vertices, for each $1 \leq k \leq \binom{n}{2}$ the algorithm repeatedly finds implications of the form $Imp_A(k, \pi)$ until the set obtained so far implies every implication of length k . To find a new implication of length k we check if there exists a permutation $\pi \in S_n$ and an $n \times n$ adjacency matrix A of Boolean variables such that C is satisfied, but there exists a lex implication $Imp_A(k, \pi)$ which is not satisfied where C denotes the conjunction of the implications selected so far. This process continues iterating for implications of all lengths, starting from short implications, $k = 1$, and finishing with the longest implications, $k = \binom{n}{2}$. In the algorithm we apply a reduce step to remove redundant implications after each increment of the value k .

Table 4 details the computation of compact complete symmetry breaks following Algorithm 3 (lex implications) and provides a comparison with those computed in [11] (canonizing), and the symmetry breaks from [12] (isolators). For

Algorithm 3. Generating Complete Implication Set

```

init:  $C \leftarrow \{ \}$ 
for  $k = 1$  to  $\binom{n}{2}$  do
  while  $\exists \pi \in S_n, G \in G_n$  s.t  $(C \wedge \neg Imp_G(k, \pi))$  is satisfiable do
     $C \leftarrow C \cup \{ Imp_G(k, \pi) \}$ 
   $C \leftarrow reduce(C)$ 
return  $C$ 

```

each value of n (number of vertices) we detail the size of the symmetry break derived and the time it took to compute it. For the symmetry breaks of [11], size is reported in the number of lex constraints (“lex”) and also in the number of clauses in their encoding to CNF. For the symmetry breaks derived in this paper, size is reported in the number of lex implications (“imp”) and also in the number of clauses in their encoding to CNF. Isolators are, by definition, sets of clauses. The times in the right column (Isolator) of the table are reported from [12]. These were obtained, for different values of n , using different techniques. Thus, the computation for 8 nodes is faster than the one for 7 nodes. For 7 nodes, Heule reports in [12], a computation involving 80,000 probes per round with 4 rounds at 7 min (average) per probe. This totals 1555 days of computation. The items denoted – indicate that the corresponding symmetry breaks cannot be computed. The only technique able to compute a symmetry break for graphs with 11 vertices is the technique presented in this paper. We note that this is the first time that a compact and complete symmetry break for graphs of size 11 has been computed.

Table 4. Computing compact and complete symmetry breaking constraints for graphs (time is in seconds except where indicated otherwise).

	Canonizing			Lex implications			Isolator from [12]	
	lex	clauses	time	imp	clauses	time	clauses	time
4	3	12	0.03	6	14	0.18	7	0.01
5	7	83	0.10	18	52	0.49	12	2.34
6	13	156	1.62	45	149	1.99	27	4.6 days
7	37	580	13.19	139	449	8.68	114	1555 days
8	135	2,724	345.37	447	1139	59.81	956	2 days
9	842	18,311	2.42 h	2,496	3736	626.20	–	–
10	7,853	250,582	93.82 h	22,542	24,512	3.10 h	–	–
11	–	–	–	274,109	277,075	8.44 days	–	–

Table 5 demonstrates the impact of having more compact complete symmetry breaks. We detail the time to compute all graphs that satisfy each form of the complete symmetry break. Because the symmetry breaks are all complete, this

number corresponds exactly to the number of non-isomorphic undirected graphs with n vertices (OEIS sequence number A000088) [21] which is detailed in the right column. We see from the table that enumerating graphs with more compact symmetry breaks significantly reduces the computation time.

5 An Application: Computing Ramsey Colorings $(4, 4; n)$

In this section we describe the impact of using compact and complete symmetry breaks. We consider a classic example of a graph search problem: the search for Ramsey graphs [23]. The graph $R(s, t; n)$ is a simple graph with n vertices, no clique of size s , and no independent set of size t . The Ramsey number $R(s, t)$ is the smallest number n for which there is no $R(s, t; n)$ graph. Table 6 reports on the search for all solutions for $R(4, 4, n)$. For $n > 17$ there are no solutions and hence the Ramsey number $R(4, 4) = 18$. The table compares three configurations: First, using the partial symmetry breaking predicate sb_ℓ^* defined in [6]. Second, using the complete canonizing symmetry breaks computed in [11]. Third, using the complete symmetry breaks computed in this paper. For each configuration we detail the size of the SAT encoding (clauses and variables), the time in seconds (except where indicated in hours) to find all solutions using a SAT solver, and the number of solutions found. The symmetry breaks based on canonizing permutations and on lex implications are both complete, so they both compute the exact number of solutions. In the upper part of the table, we use the corresponding complete symmetry breaks described in [11] (for $n \leq 10$) and in Sect. 4 of this paper (for $n \leq 11$). These symmetry breaks are “*instance independent*”. They apply to break symmetries for any graph search problem. For $12 \leq n \leq 17$ we compute “*instance dependent*” symmetry breaks. We refine Algorithm 3 to compute lex implications that break symmetries for the specific application to $R(4, 4, n)$. This is, we restrict Algorithm 3 to consider only $R(4, 4, n)$ graphs instead of all graphs in \mathcal{G}_n .

Table 5. Enumerating graphs using complete symmetry breaking methods: canonizing, lex-implications, and isolators (using Clasp where time in seconds unless indicated otherwise).

n	Canonizing	lex-imp's	Isolator [12]	Graphs
4	0.00	0.00	0.00	11
5	0.00	0.00	0.00	34
6	0.00	0.00	0.00	156
7	0.00	0.00	0.00	1,044
8	0.27	0.11	0.04	12,346
9	33.34	3.65	–	274,668
10	5.78 h	542.25	–	12,005,168
11	–	2.69 days	–	1,018,997,864

Table 6. Enumerating Ramsey graphs using \mathbf{sb}_ℓ^* , canonizing sets and lex implications. (Clasp solver, computation time in seconds).

Instance	\mathbf{sb}_ℓ^*				Canonizing sets			Lex implications			
	cls	vars	sat	sols	cls	vars	sat	cls	vars	sat	exact
$R(4, 4, 4)$	22	10	0.00	9	17	9	0.00	68	21	0.00	9
$R(4, 4, 5)$	80	24	0.00	33	235	55	0.00	208	55	0.00	24
$R(4, 4, 6)$	195	48	0.00	178	315	72	0.00	495	120	0.00	84
$R(4, 4, 7)$	390	85	0.00	1,478	1,395	286	0.00	1,049	231	0.00	362
$R(4, 4, 8)$	690	138	0.03	16,919	10,885	2,177	0.04	2,099	406	0.02	2,079
$R(4, 4, 9)$	1,122	210	0.51	227,648	89,877	17,961	1.56	5,268	666	0.23	14,701
$R(4, 4, 10)$	1,715	304	9.97	2,891,024	1,406,100	281,181	149.15	26,922	1,035	5.43	103,706
$R(4, 4, 11)$	2,500	423	428.79	25,616,963	–	–	–	280,709	1,540	726.62	546,356
$R(4, 4, 12)$	3,510	570	5561.06	107,509,048	–	–	–	48,363	2,715	129.71	1,449,390
$R(4, 4, 13)$	4,780	748	29426.23	131,638,650	–	–	–	57,747	3,751	133.64	1,184,323
$R(4, 4, 14)$	6,347	960	8325.25	21,181,746	–	–	–	36,505	5,055	31.18	130,818
$R(4, 4, 15)$	8,250	1,209	281.79	144,663	183,985	36,356	26.21	30,855	6,669	55.95	640
$R(4, 4, 16)$	10,530	1,498	14.38	94	30,890	5,570	12.34	39,131	8,638	19.43	2
$R(4, 4, 17)$	13,230	1,830	5.63	4	15,255	2,235	7.51	49,953	11,010	20.69	1

The lower part of Table 6 reports on the search for all solutions of $R(4, 4, n)$ for $n \geq 12$ using these complete symmetry breaks. The items denoted – indicate that the corresponding symmetry breaks cannot be computed within the timeout period (72 h).

It can be seen that for $12 \leq n \leq 15$, in which the number of non-isomorphic solutions is large, these problem dependent symmetry breaks significantly improve the solving time over the partial symmetry break \mathbf{sb}_ℓ^* . For $n = 16$, the symmetry break computed here is significantly stronger than \mathbf{sb}_ℓ^* , allowing for only 2 instead of 94 solutions.

Using the lex-implications approach we were able to compute instance dependent symmetry breaks for all $R(4, 4, n)$ instances whereas the computation of canonizing sets exceeded the timeout for three cases.

6 Conclusion

We provided an analysis of the redundancy in symmetry breaking constraints for graphs and matrix models. Previous work had shown that many of the lex constraints in the LEXLEADER symmetry break are redundant. Here, we considered the decomposition of lex constraints in lex implications, and showed that many of them are redundant in complete symmetry breaks. This allowed us to reduce the size of complete symmetry breaks for graphs by an order of magnitude, and enabled us to compute a complete and compact symmetry break for graphs on 11 nodes.

Furthermore, we analyzed partial symmetry breaks and the redundancies in them. While small symmetry breaks like \mathbf{sb}_ℓ for graphs, and lex^2 for matrices do

not contain any redundant lex implications, there are significant redundancies in their extensions sb_ℓ^* and lex^+ , lex^* , respectively.

References

1. Puget, J.-F.: On the satisfiability of symmetrical constrained satisfaction problems. In: Komorowski, J., Raś, Z.W. (eds.) ISMIS 1993. LNCS, vol. 689, pp. 350–361. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56804-2_33
2. Crawford, J.M., Ginsberg, M.L., Luks, E.M., Roy, A.: Symmetry-breaking predicates for search problems. In: Aiello, L.C., Doyle, J., Shapiro, S.C. (eds.): Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR 1996), Cambridge, Massachusetts, USA, 5–8 November 1996, pp. 148–159. Morgan Kaufmann (1996)
3. Shlyakhter, I.: Generating effective symmetry-breaking predicates for search problems. *Discrete Appl. Math.* **155**(12), 1539–1548 (2007)
4. Walsh, T.: General symmetry breaking constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 650–664. Springer, Heidelberg (2006). https://doi.org/10.1007/11889205_46
5. Walsh, T.: Symmetry breaking constraints: recent results. In: Hoffmann, J., Selman, B. (eds.) Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, Toronto, Ontario, Canada, 22–26 July 2012. AAAI Press (2012)
6. Codish, M., Miller, A., Prosser, P., Stuckey, P.J.: Breaking symmetries in graph representation. In: Rossi, F. (ed.) Proceedings of the 23rd International Joint Conference on Artificial Intelligence, IJCAI 2013, Beijing, China, 3–9 August 2013, pp. 510–516. IJCAI/AAAI (2013)
7. Flener, P., Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetries in matrix models. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 462–477. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46135-3_31
8. Frisch, A.M., Jefferson, C., Miguel, I.: Constraints for breaking more row and column symmetries. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 318–332. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45193-8_22
9. Grayland, A., Miguel, I., Roney-Dougal, C.M.: Snake lex: an alternative to double lex. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 391–399. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_32
10. Read, R.C.: Every one a winner or how to avoid isomorphism search when cataloguing combinatorial configurations. *Ann. Discrete Math.* **2**, 107–120 (1978)
11. Itzhakov, A., Codish, M.: Breaking symmetries in graph search with canonizing sets. *Constraints* **21**(3), 357–374 (2016)
12. Heule, M.J.H.: The quest for perfect and compact symmetry breaking for graph problems. In: Davenport, J.H., Negru, V., Ida, T., Jelebean, T., Petcu, D., Watt, S.M., Zaharie, D. (eds.) 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2016, Timisoara, Romania, 24–27 September 2016, pp. 149–156. IEEE Computer Society (2016)
13. Frisch, A.M., Harvey, W.: Constraints for breaking all row and column symmetries in a three-by-two matrix. In: Proceedings of SymCon 2003 (2003)
14. Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Propagation algorithms for lexicographic ordering constraints. *Artif. Intell.* **170**(10), 803–834 (2006)

15. Smith, B.: Symmetry breaking constraints in constraint programming (2010). Slides published online. http://ta.twi.tudelft.nl/wst/users/achill/MFOSymOpt2010/MFOSymOpt2010/Oberwolfach_Mini-Workshop_files/BarbaraMfoSlides.ppt
16. Metodi, A., Codish, M., Stuckey, P.J.: Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. *J. Artif. Intell. Res. (JAIR)* **46**, 303–341 (2013)
17. Audemard, G., Simon, L.: Glucose 4.0 SAT solver. <http://www.labri.fr/perso/lsimon/glucose/>
18. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: from theory to practice. *Artif. Intell.* **187**, 52–89 (2012)
19. Cameron, R.D., Colbourn, C.J., Read, R.C., Wormald, N.C.: Cataloguing the graphs on 10 vertices. *J. Graph Theor.* **9**(4), 551–562 (1985)
20. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. *J. Symb. Comput.* **2**(3), 293–304 (1986)
21. The on-line encyclopedia of integer sequences. Published electronically (2010). <http://oeis.org>
22. Fourdrinoy, O., Grégoire, É., Mazure, B., Saïs, L.: Eliminating redundant clauses in SAT instances. In: Van Hentenryck, P., Wolsey, L. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 71–83. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72397-4_6
23. Radziszowski, S.P.: Small Ramsey numbers. *Electron. J. Comb.* (1994). Revision 14 January 2014



Model Checking Parameterized by the Semantics in Maude

Adrián Riesco^(✉) 

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid, Madrid, Spain
ariesco@fdi.ucm.es

Abstract. Model checking is an automatic verification technique for analyzing whether some properties hold in a model. Maude is a high-performance logical framework and model checking tool where many different concurrent programming languages have been specified and analyzed. However, the counterexample generated by Maude when a property fails does not correspond to the language being specified but to the Maude rules, which makes it difficult to understand. In this paper we present two metalevel transformations for relating counterexamples and semantics when dealing with the semantics of concurrent languages, hence allowing users to model check real code while easing the interpretation of the counterexamples. These transformations can be applied to any semantics following a message-passing or a shared memory approach. These transformations have been implemented in a Maude prototype; we illustrate the tool with examples.

Keywords: Model checking · Semantics · Message passing
Shared memory · Maude

1 Introduction

Model checking [5] is an automatic technique for checking whether a property, usually stated in modal logic, holds in a system. It starts from an initial state and exhaustively traverses all the reachable states, which makes it a useful verification tool for concurrent systems, where complex interleaving failures might be overlooked during the implementation and testing phases. State-of-the-art model checkers, such as Spin [2] and NuSMV [4], allow users to analyze models of their algorithms, but they do not check the actual application code directly. For this reason, the relation between programs, models, and their corresponding translations are subjects of growing concern in the model-checking community, as shown for example by the Java PathFinder [13] community.

This research has been partially supported by the MINECO Spanish project *TRACES* (TIN2015-67522-C3-3-R) and by the Comunidad de Madrid project *N-Greens Software-CM* (S2013/ICE-2731).

Maude [6] is a high-performance logical framework where the semantics of other programming languages can be specified and analyzed. Maude modules correspond to specifications in *rewriting logic* [14], a logic that allows specifiers to represent many models of concurrent and distributed systems. This logic is an extension of *membership equational logic* [3], an equational logic that, in addition to equations, allows the statement of *membership axioms* characterizing the elements of a sort. Rewriting logic extends membership equational logic by adding rewrite rules that represent transitions in a concurrent system and can be nondeterministic. Moreover, an important feature of rewriting logic is that it is reflective, that is, it can be faithfully interpreted in terms of itself. This feature is efficiently implemented in Maude by means of the `META-LEVEL` module [6, Chap. 14], which allows us to use Maude modules and terms as usual data.

Defining the semantics of a programming language in Maude presents many advantages over other languages: first, Maude specifications are executable, so the specification gives the specifier an interpreter of the semantics for free. Moreover, Maude provides several analysis tools, including an LTL model checker. Hence, since the seminal proposal of using rewriting logic as a semantic framework in [15], Maude has been used to specify the semantics of many languages, such as LOTOS [19], CCS [19], and Java [10]. Moreover, the \mathbb{K} -Maude compiler [18], which is able to translate \mathbb{K} [17] specifications into Maude, has eased the methodology to describe programming language semantics in Maude, as shown e.g. by the C semantics in [9]. However, when using the model checker for checking properties on programs whose semantics have been defined in Maude, we obtain a counterexample that refers to the semantics of the language but not directly to the actual program under analysis. Given the complex nature of concurrent systems, this extra layer of complexity makes the counterexample even more difficult to understand in real applications, preventing specifiers from understanding the error and being able to fix it.

We present in this paper two generic transformations, implemented using Maude metalevel, for relating the counterexample generated by the Maude model checker with the semantics of the language being executed. These transformations can be applied to concurrent programs following either a message-passing approach or a shared-memory approach. They reduce the counterexample, focus on the main events depending on the semantics, and return a JSON-like¹ result that is easy to follow and manipulate later, in the sense that it can be parsed in an automatic way by other applications and programming languages like Python in order to perform other analyses. In this way Maude specifiers get, in addition to an interpreter for their language, a model checker for the object language for free. Moreover, we also get a model checker for real code for all those programming languages included that are already specified in Maude. To the best of our knowledge, this kind of generic, metalevel transformation is novel in model checking.

The rest of the paper is organized as follows: Sect. 2 introduces the different types of semantics discussed throughout the rest of the paper. Section 3 presents

¹ See <http://www.json.org/> for details.

the transformation for languages based on shared memory, while Sect. 4 presents the transformation for message-passing style. Finally, Sect. 5 concludes and outlines some lines of future work. The code of the tool, examples, and more information is available at <https://github.com/ariesco/MCPS>.

2 Preliminaries

We present in this section how the semantics for message-passing and shared-memory programming languages can be specified in Maude. We present one example for each semantics and show how it is model-checked; we will show in the next sections how the results obtained from these analyses are transformed. Note that the semantics here are just simple examples illustrating the power of the tool; it can be applied to any other semantics following the same principles.

2.1 Implementing Semantics in Maude

Semantics are represented in Maude by means of conditional *rewrite rules*, that stand for transitions between states. In this way, each inference rule of the form:

$$\frac{P_1 \dots P_n}{state_1 \Rightarrow state_2} id$$

in the semantics, which indicates that $state_2$ is reached from $state_1$ if the premises $P_1 \dots P_n$ hold, is written in Maude as:

```
cr1 [id] : state1 => state2 if P1 /\ ... /\ Pn .
```

where the conditions P_i can be either equalities (possibly involving some auxiliary functions), that will be solved by applying equations, or rewrite conditions, that indicate that some extra transitions must hold.

In the following we will give the intuitive ideas underlying the syntax of our languages and limit Maude code to some rewrite rules defining the language semantics, so the ideas can be followed by non-experts. Type definitions, auxiliary functions, and much more information is available in the repository above.

2.2 Shared-Memory Semantics

We use a modification of the imperative language in [6, Chap. 13] as running example. This language includes assignments ($X := E$), sequential composition ($INS ; INS'$), conditional statements (**if** COND **then** INS **fi**), and loops (**while** COND **do** INS **od** and **repeat** INS **forever**), for X a variable, E an expression, INS and INS' sequences of instructions, and COND a condition. Processes executing programs written with this syntax are wrapped into processes of the form $[ID, P]$, with ID a natural number standing for the process identifier and P the program being executed. Finally, the whole system is a pair of the form $[PS, M]$, with PS a set of processes (put together by using l) and M

```

repeat
  c1 := 1 ; *** It should be c2 := 1 ;
  while c1 = 1 do
    if turn = 1 then
      c2 := 0 ;
      while turn = 1 do skip od ;
      c2 := 1
    fi
  od ;
  cs2 := 1 ; *** start critical section for process 2
  cs2 := 0 ; *** end critical section for process 2
  turn := 1 ;
  c2 := 0
forever
    
```

Fig. 1. Simplified Dekker algorithm

the memory, which consists of a set of pairs $[V, N]$, with V a variable and N a natural number. We assume all variables in the system are initialized beforehand.

The semantics of this system are defined by using rewrite rules for each instruction. For example, the rule `asg` indicates that, given a process I where the first instruction to be executed is the assignment $Q := N$ (the variable S stands for the rest of processes) and the memory contains the pair $[Q, X]$ (M stands for the rest of the pairs in the memory), the instruction is executed by updating the value of the variable from X to N .²

$$\begin{array}{l}
 \text{rl [asg]} : \{[I, Q := N ; R] \mid S, [Q, X] M\} \\
 \Rightarrow \quad \{[I, R] \quad \quad \quad \mid S, [Q, N] M\} .
 \end{array}$$

Similarly, a `repeat` puts the body of the loop before repeating the instruction:

$$\begin{array}{l}
 \text{rl [repeat]} : \{[I, \text{repeat } P \text{ forever} ; R] \mid S, M\} \\
 \Rightarrow \quad \{[I, P ; \text{repeat } P \text{ forever} ; R] \mid S, M\} .
 \end{array}$$

Using this syntax, [6, Chap. 13] describes the verification of the Dekker algorithm, a well-known protocol for ensuring mutual exclusion where each process actively waits for its turn; this turn is indicated by a variable that is only changed by the process exiting the critical section. We present in Fig. 1 a simplification of the algorithm for the second process (the first is defined analogously by changing the variables $c1/c2$, $cs1/cs2$, and the value in `turn`) where a bug has been introduced, hence violating the mutual exclusion property. Note that the value of `csi` is used to determine if process i is in the critical section.

Hence, given (1) the initial state $\{[1, p1] \mid [2, p2], [c1, 0][c2, 0][cs1, 0][cs2, 0][turn, 1]\}$, with $p1$ and $p2$ the corresponding version of

² Note that this is a small-step semantics and hence N is completely evaluated. In general we may need to compute the expression on the righthand side in a rewrite condition.


```

red modelCheck(initial, []~ (enterCrit(cs1) /\ enterCrit(cs2))) .
result ModelCheckResult: counterexample({{[1,repeat c1 := 1 ;
while c2 = 1 do if turn = 2 then c1 := 0 ; while turn = 2 do skip
od ; c1 := 1 fi od ; cs1 := 1 ; cs1 := 0 ; turn := 2 ; c1 := 0 forever]
| [2,repeat c1 := 1 ; while c1 = 1 do if turn = 1 then c2 := 0 ;
while turn = 1 do skip od ; c2 := 1 fi od ; cs2 := 1 ; cs2 := 0 ;
turn := 1 ; c2 := 0 forever],[c1,0] [c2,0] [cs1,0] [cs2,0] [turn,1]},
repeat}
{{[1,c1 := 1 ; while c2 = 1 do if turn = 2 then c1 := 0 ; while turn = 2
do skip od ; c1 := 1 fi od ; cs1 := 1 ; cs1 := 0 ; turn := 2 ; c1 := 0 ;
repeat c1 := 1 ; while c2 = 1 do if turn = 2 then c1 := 0 ; while turn = 2
do skip od ; c1 := 1 fi od ; cs1 := 1 ; cs1 := 0 ; turn := 2 ; c1 := 0
forever] | [2,repeat c1 := 1 ; while c1 = 1 do if turn = 1 then c2 := 0 ;
while turn = 1 do skip od ; c2 := 1 fi od ; cs2 := 1 ; cs2 := 0 ; turn := 1
; c2 := 0 forever],[c1,0] [c2,0] [cs1,0] [cs2,0] [turn,1]},'asg', ...)}

```

Fig. 2. Counterexample fragment for mutual exclusion in the (buggy) Dekker algorithm

the Dekker algorithm for the first and the second process, respectively, and (2) an atomic formula `enterCrit` that holds when the variable given as argument (either `cs1` or `cs2`) has value 1 (i.e., the corresponding process is in the critical section), we check whether mutual exclusion holds in Fig. 2, where we have highlighted the command (first rectangle) and the result (second and third rectangles). As shown in the figure, the property does not hold and hence a counterexample is returned; it consists of a list of pairs containing a state and the identifier of the rule used to reach the next state. In our case we just depict the first two pairs; the first one consists of the initial state and the rule label `repeat` (first inner rectangle), indicating that this rule is applied to reach the state in the second pair, where in turn `asg` will be used (second inner rectangle). This counterexample is difficult to follow not only because of the presentation, it also gives the user information that it is useful from the Maude point of view but not from the programming language point of view. For example, the user might not be interested in the steps involving the `repeat` rule, since it does not modify the memory. We will show in Sect. 3 how this counterexample is transformed.

2.3 Message-Passing Semantics

We consider for our message-passing semantics the simple functional language in [19],³ which supports `let` and conditional expressions, as well as basic arithmetic and Boolean operations. First, we expand it with expressions of the form `to ID : M` for sending messages, with both `ID` and `M` natural numbers standing for the identifier of the addressee and the message, respectively, and `receive` expressions for receiving them. Then, we define processes as terms of the form `[ID | E | ML]`, with `ID` a natural number identifying the process, `E` the expression being evaluated in the process, and `ML` a list of natural numbers standing for

³ For the sake of conciseness we use syntactic sugar for numbers and variables.

the messages received thus far (we consider the head of the list is the leftmost element). Finally, the whole system is represented as a term of the form $|| \text{PS}, \text{D} ||$, for PS a set of processes and D a set of function declarations.

In this language we have rules of the form $\text{D}, \text{ro} \vdash e \Rightarrow e'$ for simplifying expressions, given a set of declarations D , an environment ro , and expressions e and e' . For example, rule **Let1** below shows how the expression e in a **let** expression is simplified by applying a rewrite condition that computes e'' . On the other hand, rule **Let2** just applies the appropriate substitution when a value has been obtained for the variable:

```
cr1 [Let1] : D,ro ⊢ let x = e in e' => let x = e'' in e'
if D,ro ⊢ e => e'' .
```

```
rl [Let2] : D,ro ⊢ let x = v in e' => e'[v / x] .
```

Similarly, we need rules at the process level to model how messages are sent and received. Rule **send** below shows how a message being processed by id and addressed to id' is introduced into the list of received messages of id' , while value 1 is used in id to indicate that the message was delivered correctly. Rule **receive** is in charge of consuming messages: it substitutes a **receive** expression by the first message in the list:

```
rl [send] :
  || [id | let x = (to id' : n) in e | nl] [id' | e' | nl'] ps , D ||
=> || [id | let x = 1 in e | nl] [id' | e' | nl' . n] ps , D || .
```

```
rl [receive] :
  [ id | let x = receive in e | n . nl ] => [ id | let x = n in e | nl ] .
```

We will use a simple synchronization protocol between a server and two clients to illustrate how the model checker behaves in this case. Hence, we have the following initial state, with the server identified by 0 and the clients by 1 and 2. Note that the server receives the client identifiers as arguments, while the clients receive the server identifier:

```
|| [0 | server(1, 2) | nilML] [1 | client(0) | nilML]
   [2 | client(0) | nilML], decs ||
```

The declarations **decs**, shown below, indicate that the server sends a message (0) to the process identified by the first argument (client 1 in this case), another message (1) to the process identified by the second argument (2), then waits for two messages and returns 1 if it receives 0 and 1 (in this order) and 0 otherwise. In turn, the client receives a message and just returns the same message to the server, whose identifier received as parameter:

```

reduce in TEST : modelCheck(init, <> [] finalValue(0, 1)) .
result ModelCheckResult: counterexample({| [0 | server(1,2) | nilML]
[1 | client(0) | nilML] [2 | client(0) | nilML], client(x) <= let y =
receive in let z = to x : y in z & server(x,y) <= let a = to x : 0 in
let b = to y : 1 in let c = receive in let d = receive in If Equal(c, 0)
And Equal(d, 1) Then 1 Else 0 |,'advance-process})
{| [0 | let a = to 1 : 0 in let b = to 2 : 1 in let c = receive in
let d = receive in If Equal(c, 0) And Equal(d, 1) Then 1 Else 0 | nilML]
[1 | client(0) | nilML] [2 | client(0) | nilML], client(x)<= let y =
receive in let z = to x : y in z & server(x,y)<= let a = to x : 0 in
let b = to y : 1 in let c = receive in let d = receive in If Equal(c, 0)
And Equal(d, 1) Then 1 Else 0 |,'send', ...}

```

Fig. 3. Counterexample for message-passing semantics

```

server(x, y) <= let a = to x : 0
                in let b = to y : 1
                  in let c = receive
                    in let d = receive
                      in If Equal(c, 0) And Equal(d, 1)
                        Then 1 Else 0 &
client(x)      <= let y = receive
                  in let z = to x : y in z

```

A naïve user might expect messages from clients to be received in the same order as they were sent from the server, and hence the final state to be always 1. Figure 3 shows the command (first rectangle) and the first two states of the counterexample for this property (second and third rectangles, respectively), where `finalValue` is an atomic proposition that holds if the process identified by the first argument contains the expression given as second argument. The first step just substitutes the function call by the body of the function, while the second one is in charge of sending the first message in the server, as highlighted by the inner rectangle. We will see in Sect. 4 how to improve this trace.

2.4 Maude Metalevel and Loop Mode

The transformations presented in this paper have been implemented in an interactive Maude tool extending Full Maude [6, Part II] and using Maude metalevel capabilities [6, Chap. 14].

Full Maude is an extension of Maude written in Maude itself. It provides an input/output loop, an explicit state, and facilities to define, parse, and execute new commands, making it the most appropriate option to develop interactive Maude applications. It is worth noting that commands in Full Maude must be enclosed in parentheses, as required by the Loop Maude [6, Chap. 17], the built-in Maude module in charge of dealing with input/output information. For this reason, all commands in Sects. 3 and 4 will follow this convention.

On the other hand, Maude metalevel allows users to use Maude modules and terms as usual data. This feature allows us to:

- Traverse modules and identify those rules modifying terms of a given sort (e.g. the memory) or creating/consuming terms built with particular operators (e.g. messages).
- Identify the subterms involved in each step. This analysis is twofold: (i) given the whole state, we are interested in identifying the particular subterm being rewritten (e.g. identify the process executing the code among the set of processes), and (ii) recognize particular parts of the subterm found in the previous step to isolate elements of interest (e.g. messages).
- Manipulate the counterexample obtained when model checking a system. In particular, we can use the information obtained when traversing the module to prune the counterexample and the information about subterms to distinguish among the different parts of each state (e.g. memory, processes, and messages).

3 Model Checking Shared-Memory Languages

In this section we present the transformation used for programming languages following a shared-memory approach. In this transformation we rely in the following assumption: *properties refer to memory states*. Hence, we only need to keep those transitions in the original counterexample performed by rules that modify the memory. For example, for the program in Sect. 2.1 we will only keep those steps involving the `asg` rule. We consider this is a safe assumption, since in these systems the access to the shared resources is critical.

Once we have decided the transitions that we want to keep, we must decide how to display each step. We decided to follow a JSON-like format and display the following information:

- The process executed (field `unit`) when the rule is applied. If the process has an identifier it will be displayed in the `id` field.
- The whole system (field `system`) before the rewrite rule is applied.
- The state of the memory. Since the memory will be modified by the application of the rule, we present the state *before* applying the rule (field `memory-before`) and *after* applying it (field `memory-after`). In this way the user can inspect the effects of the rule. Note that this field is a list, since in general different types of memory can be used.
- Since we can work at the metalevel, we decided to display the value of all atomic formulas before and after applying the rule, so the user can understand the values taken by the LTL formula (field `props`). For each atomic proposition in the formula we display its name, arguments, and how its value changed when the current rule is applied.

Algorithm 1 presents the transformation for shared memory, where all functions but `head` and `tail` are implemented at the metalevel, since they manipulate

Data: Counterexample c , semantics \mathcal{S} , sorts ms for memory terms, sort p for processes, atomic propositions aps , and (optionally) id argument.

Result: Transformed counterexample.

```

rule_labels = memoryRules( $\mathcal{S}$ );
 $c$  = close( $\mathcal{S}$ ,  $c$ , rule_labels);
while not empty( $c$ ) do
  ( $term$ ,  $label$ ) = head( $c$ );
   $c$  = tail( $c$ );
  if  $label \in rule\_labels$  then
    ( $term'$ ,  $label'$ ) = head( $c$ );
     $sub$  = match( $\mathcal{S}$ ,  $term$ ,  $label$ ,  $term'$ );
     $lhs$  = apply( $sub$ , getLefthandSide( $\mathcal{S}$ ,  $label$ ));
     $m\_info$  = getMemoryInfo( $term$ ,  $term'$ ,  $ms$ );
     $s\_info$  = getStateInfo( $term$ ,  $ms$ );
     $p\_info$  = getProcessInfo( $lhs$ ,  $p$ , [ $id$ ]);
     $props\_info$  = getPropsInfo( $term$ ,  $term'$ ,  $aps$ );
    display( $m\_info$ ,  $s\_info$ ,  $p\_info$ ,  $props\_info$ );
  end
end
end

```

Algorithm 1. Transformation for shared-memory semantics

modules, rules, and terms. We first extract from the semantics those rules that modify the memory by using `memoryRules`. Then, we make sure the last transition in the counterexample does not use a rule in this set; if this is the case, the function `close` explicitly adds the next state⁴ and uses a special label not in `rule_labels` to make sure the condition in the `while` loop skips it. Then the loop traverses all the states in the counterexample; when we find a step whose label is in `rule_labels` then we take the next state to find the matching (function `match`) that was used in the rule. This is required because, given a term and a rule, many different matchings are possible, so we need to ensure that we use the correct one. Then, we apply this matching to instantiate the lefthand side of the rule being used, hence obtaining lhs (function `apply`). This subterm is the one containing the information about the process being executed, while the term in the counterexample ($term$) contains the information about the whole system. We use appropriate pretty-printing functions to display the corresponding information.

The current version of the system cannot infer the sort for the memory or the sort for the processes (that we call *units*). Hence, we require the user to introduce these sorts. In our example, we would start by introducing `Memory` as the sort used for the memory and `Process` as the sort used for processes. Moreover, we indicate that the first argument for `Process` stands for the identifier:⁵

⁴ Since a cycle is required to evaluate an LTL formula, this new state has appeared before in the counterexample and there is no need to explore it again.

⁵ If the constructor does not include an identifier we would use `(unit Process .)`.

```

Maude> (memory sorts Memory .)
Memory sorts introduced: Memory

Maude> (unit Process id 1 .)
Unit sort introduced: Process
It is identified by the 1 argument.

Maude> (shared memory analysis modelCheck(initial,
      []~ (enterCrit(cs1) /\ enterCrit(cs2))) .)
...
{ id = 1,
  unit = ...,
  system = ...,
  memory-before = {[c1,1][c2,0][cs1,0][cs2,1][turn,1]},
  memory-after  = {[c1,1][c2,0][cs1,1][cs2,1][turn,1]},
  props = [{name = enterCrit,
            args =[cs1],
            prop = false -> true},
           {name = enterCrit,
            args =[cs2],
            prop = true -> true}]
} ...

```

Fig. 4. Transformed counterexample for shared-memory semantics

Note that the first command allows the user to introduce several sorts for the memory, since different representations can be used for registers, main memory, etc. Once this information has been introduced into the system, it infers that the single rule modifying the memory is `asg`, so only the steps using this rule in the counterexample will be displayed. Now, we can execute the `shared memory analysis` command with the same model-checking command that we used in Sect. 2.2. We present a fragment of the transformed trace in Fig. 4, where `unit` and `system` are not shown for the sake of readability. The step shown in the figure corresponds to the rewrite step that violates mutual exclusion: the process identified by 1 is executed and it goes into the critical section (variable `cs1` changes its value from 0 to 1, as we have highlighted in the figure), satisfying the corresponding property (`enterCrit(cs1)`); since the second process was into the critical section as well (as we can see by checking `cs2` or the corresponding property), the formula fails. However, we also notice that the process 1 behaved appropriately, since it was its turn (see variable `turn`), so we would inspect the trace for the second process to find the error.

Hence, the trace now can be read more easily, it contains less states (while the original counterexample had 89 states, the transformed one has 58), and it is displayed in a format that can be parsed and analyzed later if required.

4 Model Checking Message-Passing Languages

We present in this section two different ways to transform counterexamples like the one shown in Sect. 2.3, so the Maude semantics become transparent to the user. While the first one summarizes the actions performed by the processes during the computation, the second one presents trace-like information with the main actions that took place.

We denote as **summary** mode our first approach, which presents the expression reached in each process, as well as the sent and consumed messages. In order to do so, we need the user to introduce the sort for the processes (and the argument standing for its identifier, if it exists) and the constructors for sending and consuming messages. The tool will use this information to identify those rules in charge of dealing with messages and to locate the processes and their identifiers, as well as the messages sent and consumed, so they can be displayed. Hence, this transformation presents the following information for each process:

- Its identifier (**id** field).
- Its final value (**value** field). Note that in some cases this value will not be a normal form, since some functions (e.g. servers) might be non-terminating.
- The list of messages it has sent (**sent** field).
- The list of messages it has consumed (**consumed** field).

However, in some cases it is also useful to understand the interleaving between different messages and processes. For this reason, we decided to present a trace-like counterexample, that we call **trace** mode. However, in this semantics is not clear the notion of “step,” so we first decided to focus on messages and display information when a message is sent or consumed. Then, we noticed that some properties might change some steps after a message was sent or received, and hence we decided to include in the trace those steps where at least one atomic property changes its truth value. As explained in the previous section, this information is used by executing at the metalevel all the atomic properties in the state reached in the corresponding state. In this approach each step contains the following information:

- The identifier of the process that performed the action (**id** field).
- The action that took place (**action** field), which can be either **msg-consumed**, **msg-sent**, and **prop-changed**, which stand for messages consumed, messages sent, and truth value of atomic propositions changed, respectively.
- The messages involved in the action (**messages** field). This field is omitted when the action is not referred to message creation or consumption.
- The state of all processes before and after applying the rule (**processes-before** and **processes-after** fields, respectively).
- How the properties changed with the rewrite rule (**props** field), which are displayed as explained in the previous section for shared memory.

Algorithm 2 presents this transformation, where again all functions but **head** and **tail** must be implemented at the metalevel. It first analyzes the semantics to

extract those rules in charge of sending and consuming messages, respectively. This step requires the function to traverse all rules and choose those whose righthand side either creates terms built with operators in *os* or removes terms built with *oc* (both actions with respect to the lefthand side). As explained in the previous section, we use the function `close` to add an extra final state if needed (i.e., if a message is involved or the properties change), since we need pairs of states to infer the matching. Then, we initialize the list of processes and start the loop: if the current label involves messages then we compute the substitution and the lefthand side of the rule as we did in the previous section and distinguish whether the event consisted of sending or consuming messages. We use the appropriate operators (either *os* or *oc*) to obtain the current process, the messages, and the event that took place. Note that this inference is more complex than the one for shared memory, since in the case of synchronous communication many processes might appear in the rule and we must select the one being executed, that is, containing terms built with the operators *os* or *oc*. Finally, we update the appropriate process in the list (if it did not exist a new process with that identifier is created) with the `update` function and the information is displayed depending on the selected mode.

In our example, we would indicate that processes are terms of sort `Process` and their identifier is its first argument. Similarly, we would state `to:_ .` as the instruction for sending messages and `receive` for the one consuming them:

```
Maude> (unit Process id 1 .)
Unit sort introduced: Process
It is identified by the 1 argument.

Maude> (msg creation to:_ .)
Message creation operators introduced: to:_
Maude> (msg consumption receive .)
Message consumption operators introduced: receive
```

Similarly, if we want a summary of the execution we would set the mode to `summary` (which is the default one) as follows:

```
Maude> (set mode summary .)
Mode summary selected.
```

Figure 5 shows the result when using this transformation to the example in Sect. 2.3. We see that the server, identified by 0, finished with value 0 after consuming the messages in the order 1, 0, while the clients finished as expected. With this information the user realizes that a different interleaving is possible and can fix its program. On the other hand, to use the `trace` mode in our example we would use the following command:

```
Maude> (set mode trace .)
Mode trace selected.
```

Figure 6 shows the instant when process 0 (the server) consumes the message from client 1. We have omitted processes 1 and 2 for the sake of readability,

Data: Counterexample c , semantics \mathcal{S} , operators os for sending messages, operators oc for consuming messages, sort p for processes, atomic propositions aps , and (optionally) id argument.

Result: Transformed counterexample.

```

send_labels = sendRules( $\mathcal{S}$ ,  $os$ );
cons_labels = consumeRules( $\mathcal{S}$ ,  $oc$ );
 $c$  = close( $\mathcal{S}$ ,  $c$ , rule_labels);
proc_list = [];
while ( $|c| > 1$ ) do
  ( $term$ ,  $label$ ) = head( $c$ );
   $c$  = tail( $c$ );
  ( $term'$ ,  $label'$ ) = head( $c$ );
   $p\_info$  = getPropsInfo( $term$ ,  $term'$ ,  $aps$ );
  if  $label \in (send\_labels \cup cons\_labels)$  then
     $sub$  = match( $\mathcal{S}$ ,  $term$ ,  $label$ ,  $term'$ );
     $lhs$  = apply( $sub$ , getLefthandSide( $\mathcal{S}$ ,  $label$ ));
    if  $label \in send\_labels$  then
       $p\_info$  = getProcessInfo( $lhs$ ,  $p$ , [ $id$ ],  $os$ );
       $msgs$  = getMsgProcessed( $lhs$ ,  $os$ );
       $event$  = send;
    else
       $p\_info$  = getProcessInfo( $lhs$ ,  $p$ , [ $id$ ],  $cs$ );
       $msgs$  = getMsgProcessed( $lhs$ ,  $cs$ );
       $event$  = consume;
    end
     $proc\_list[p\_info]$  = update( $proc\_list$ ,  $p\_info$ ,  $event$ ,  $msgs$ );
    display( $proc\_list$ ,  $p\_info$ ,  $event$ ,  $msgs$ ,  $p\_info$ ) ; // only in trace mode
  end
  else if changed( $p\_info$ ) then
    | display( $term$ ,  $term'$ ,  $p\_info$ ) ; // only in trace mode
  end
end
display( $proc\_list$ ) ; // only in summary mode

```

Algorithm 2. Transformation for message-passing semantics

while changes have been highlighted. Note that this message is the first one consumed by the server (in the state before the rule the list of consumed messages) because they have arrived in this order (the third argument of the `value` field, the ordered list of messages, has value `1 . 0`); after applying the rule the message has disappeared from the list, message `1` appears in the list of consumed messages, and the first `receive` in the state has been replaced by `1`. Once the user realizes this behavior was expected, he/she should change the program to take this interleaving into account.

Finally, it is worth noting that, in addition to improving the readability and providing a friendly representation, the number of steps in `trace` mode is reduced from 24 in the original counterexample to 8.

```
Maude> (msg passing analysis modelCheck(init, <> [] finalValue(0, 1)) .)

{processes = [
  { id = 0,
    value = [0 | 0 | nilML],
    sent = [to 1 : 0, to 2 : 1],
    consumed = [1, 0]},
  { id = 1,
    value = [1 | 1 | nilML],
    sent = [to 0 : 0],
    consumed = [0]},
  { id = 2,
    value = [2 | 1 | nilML],
    sent = [to 0 : 0],
    consumed = [1]}
]
```

Fig. 5. Final state for message-passing semantics

```
{id = 0,
 action = msg-consumed,
 messages = [1],
 processes-before = [
  { id = 0,
    value = [0 | let c = receive in let d = receive
              in If Equal(c, 0) And Equal(d,1) Then 1 Else 0 | 1 . 0],
    sent = [to 1 : 0, to 2 : 1],
    consumed = [], ...],
  processes-after = [
    { id = 0,
      value = [0 | let c = 1 in let d = receive
                  in If Equal(c, 0) And Equal(d,1) Then 1 Else 0 | 0],
      sent = [to 1 : 0, to 2 : 1],
      consumed = [1], ...],
    props = {name = finalValue,
              args = [0, 1],
              prop = false -> false}]
  ]
}
```

Fig. 6. Trace-like representation: message consumption

5 Concluding Remarks and Ongoing Work

In this paper we have presented two transformations that allow specifiers to model check real code and interpret the counterexamples obtained. These transformations are restricted to languages following either a shared-memory or a message passing approach. They have been implemented using Maude metalevel and are available online. To the best of our knowledge this is the first generic transformation that allows users to model check real code based on its semantics. Hence, this tool sets the basis for further development in this direction.

On the theoretical side, it is interesting to study how this approach relates to similar approaches, like the partial evaluation transformations in [7, 12].

On the tool side, it would be interesting to define transformations for other approaches, in particular for hybrid ones implementing both shared memory and message passing. We are also interested in performing a pre-analysis of the semantics to infer information about the language and hence save time and work to the user. Notably, following the analyses proposed in [16] it would be possible to identify the sorts for the memory.

Then, it would be interesting to see whether the current transformations can be improved. In [1] the authors use *slicing*, a technique to keep only those instructions related to the values reached by a set of variable of interest, to reduce the size of Maude traces. When applying this technique we face again the problems outlined in the introduction, since it works on *Maude variables* but we need it to work on *program variables*, which depend on the semantics. We would need to follow the ideas in [16] to obtain the desired result.

Regarding efficiency, following the ideas in [11] it is possible to reduce the number of states when model checking Maude specifications, hence avoiding the state-space explosion problem, by transforming rules (that generate transitions when model checking a system) into equations (that do not generate transitions) if some properties hold. These properties are the executability requirements (termination, confluence, and coherence), which can be proved in some cases using the Maude Formal Environment [8], and *invisibility*, which requires that the transformed rules do not change the truth value of the predicates. Hence, in our shared-memory model we would transform all those rules that do not modify the memory; further assumptions on the message-passing approach would be required to ensure soundness.

Overall, our long-term goal is to obtain a parameterized transformation for real languages, in the same way as Java PathFinder [13] works for Java. In this sense we will probably need to generalize other aspects of the tool, so it deals with structures such as objects.

References

1. Alpuente, M., Ballis, D., Espert, J., Romero, D.: Backward trace slicing for rewriting logic theories. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 34–48. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_5
2. Ben-Ari, M.: Principles of the Spin Model Checker. Springer, London (2008). <https://doi.org/10.1007/978-1-84628-770-1>
3. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. *Theor. Comput. Sci.* **236**, 35–132 (2000)
4. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)

6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). <https://doi.org/10.1007/978-3-540-71999-1>
7. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Semantics-based generation of verification conditions via program specialization. *Sci. Comput. Program.* **147**, 78–108 (2017)
8. Durán, F., Rocha, C., Álvarez, J.M.: Towards a Maude formal environment. In: Agha, G., Danvy, O., Meseguer, J. (eds.) *Formal Modeling: Actors, Open Systems, Biological Systems*. LNCS, vol. 7000, pp. 329–351. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_17
9. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: *Proceedings of the 39th Symposium on Principles of Programming Languages, POPL 2012*, pp. 533–544. ACM (2012)
10. Farzan, A., Chen, F., Meseguer, J., Roşu, G.: Formal analysis of Java programs in JavaFAN. In: Alur, R., Peled, D.A. (eds.) *CAV 2004*. LNCS, vol. 3114, pp. 501–505. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_46
11. Farzan, A., Meseguer, J.: State space reduction of rewrite theories using invisible transitions. In: Johnson, M., Vene, V. (eds.) *AMAST 2006*. LNCS, vol. 4019, pp. 142–157. Springer, Heidelberg (2006). https://doi.org/10.1007/11784180_13
12. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Test case generation for object-oriented imperative languages in CLP. *Theor. Pract. Log. Program.* **10**(4–6), 659–674 (2010)
13. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. *Int. J. Softw. Tools Technol. Transf.* **2**(4), 366–381 (2000)
14. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.* **96**(1), 73–155 (1992)
15. Meseguer, J., Roşu, G.: The rewriting logic semantics project. *Theor. Comput. Sci.* **373**(3), 213–237 (2007)
16. Riesco, A., Asăvoae, I.M., Asăvoae, M.: Slicing from formal semantics: Chisel. In: Huisman, M., Rubin, J. (eds.) *FASE 2017*. LNCS, vol. 10202, pp. 374–378. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_21
17. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. *J. Log. Algebr. Program.* **79**(6), 397–434 (2010)
18. Rusu, V., Lucanu, D., Serbanuta, T., Arusoaie, A., Stefanescu, A., Roşu, G.: Language definitions as rewrite theories. *J. Log. Algebraic Methods Program.* **85**(1), 98–120 (2016)
19. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. *J. Log. Algebr. Program.* **67**, 226–293 (2006)



Automated Amortised Resource Analysis for Term Rewrite Systems

Georg Moser¹(✉)  and Manuel Schneckentreither²(✉) 

¹ Department of Computer Science, University of Innsbruck, Innsbruck, Austria
georg.moser@uibk.ac.at

² Department of Information Systems, Production and Logistics Management,
University of Innsbruck, Innsbruck, Austria
manuel.schneckentreither@uibk.ac.at

Abstract. Based on earlier work on amortised resource analysis, we establish a novel automated amortised resource analysis for term rewrite systems. The method is presented in an inference system akin to a type system and gives rise to polynomial bounds on the innermost runtime complexity of the analysed term rewrite system. Our analysis does not restrict the input rewrite system in any way. This facilitates integration in a general framework for resource analysis of programs. In particular, we have implemented the method and integrated it into our tool \mathcal{TCT} .

Keywords: Analysis of algorithms · Amortised complexity
Term rewriting · Types · Automation

1 Introduction

Amortised resource analysis [1, 2] is a powerful method to assess the overall complexity of a sequence of operations precisely. It has been established by Sleator and Tarjan in the context of self-balancing data structures, which sometimes require costly operations that however balance out in the long run.

For automated resource analysis, amortised cost analysis has been in particular pioneered by Hoffmann et al., whose RaML prototype has grown into a highly sophisticated analysis tool for (higher-order) functional programs, cf. [3]. In a similar spirit, resource analysis tools for imperative programs like COSTA [4], CoFloCo [5] and LOOPUS [6] have integrated amortised reasoning. In this paper, we establish a novel automated amortised resource analysis for term rewrite systems (TRSs for short).

Consider the rewrite system \mathcal{R}_1 in Fig. 1 encoding a variant of an example by Okasaki [7, Sect. 5.2] (see also [8, Example 1]); \mathcal{R}_1 encodes an efficient implementation of a queue in functional programming. A queue is represented as a pair of two lists $\text{que}(f, r)$, encoding the initial part f and the reversal of the

This research is partly supported by DARPA/AFRL contract number FA8750-17-C-088.

1: $\text{chk}(\text{que}(\text{nil}, r)) \rightarrow \text{que}(\text{rev}(r), \text{nil})$	7: $\text{enq}(0) \rightarrow \text{que}(\text{nil}, \text{nil})$
2: $\text{chk}(\text{que}(x \# xs, r)) \rightarrow \text{que}(x \# xs, r)$	8: $\text{rev}'(\text{nil}, ys) \rightarrow ys$
3: $\text{tl}(\text{que}(x \# f, r)) \rightarrow \text{chk}(\text{que}(f, r))$	9: $\text{rev}(xs) \rightarrow \text{rev}'(xs, \text{nil})$
4: $\text{snoc}(\text{que}(f, r), x) \rightarrow \text{chk}(\text{que}(f, x \# r))$	10: $\text{hd}(\text{que}(x \# f, r)) \rightarrow x$
5: $\text{rev}'(x \# xs, ys) \rightarrow \text{rev}'(xs, x \# ys)$	11: $\text{hd}(\text{que}(\text{nil}, r)) \rightarrow \text{err_head}$
6: $\text{enq}(s(n)) \rightarrow \text{snoc}(\text{enq}(n), n)$	12: $\text{tl}(\text{que}(\text{nil}, r)) \rightarrow \text{err_tail}$

Fig. 1. Queues in rewriting

remainder r . The invariant of the algorithm is that the first list never becomes empty, which is achieved by reversing r if necessary. Should the invariant ever be violated, an exception (`err_head` or `err_tail`) is raised. To exemplify the physicist’s method of amortised analysis [2] we assign to every queue $\text{que}(f, r)$ the length of r as *potential*. Then the amortised cost for each operation is constant, as the costly reversal operation is only executed if the potential can pay for the operation, cf. [7]. Thus, based on an amortised analysis, we may deduce the optimal linear runtime complexity for \mathcal{R} .

Taking inspirations from [8,9], the amortised analysis is based on the potential method, as exemplified above. It employs the standard (small-step) semantics of innermost rewriting and exploits a *footprint* relation in order to facilitate the extension to TRSs. For the latter, we suit a corresponding notion of Avanzini and Lago [10] to our context. Due to the small-step semantics we immediately obtain an analysis which does not presuppose termination. The incorporation of the footprint relations allows the immediate adaption of the proposed method to general rule-based languages. The most significant extension, however, is the extension to standard TRSs. TRSs form a universal model of computation that underlies much of declarative programming. In the context of functional programming, TRSs form a natural abstraction of strictly typed programming languages like RaML, but natively form foundations of non-strict languages and non-typed languages as well.

Our interest in an amortised analysis for TRSs is motivated by the use of TRSs as abstract program representation within our uniform resource analyse tool TCT [11]. Incorporating a transformational approach the latter provides a state-of-the-art tool for the resource analysis of pure OCaml programs, but more generally allows the analysis of general programs. In this spirit we aim at an amortised resource for TRSs in its standard form: untyped, not necessarily left-linear, confluent, or constructor-based. Technically, the main contributions of the paper are as follows.

- Employing the standard rewriting semantics in the context of amortised resource analysis. This standardises the results and simplifies the presentations contrasted to related results on amortised analysis of TRSs cf. [8,12]. We emphasise that our analysis does not presuppose termination.
- We overcome earlier restrictions to sorted, completely defined, orthogonal and constructor TRSs, that is, we establish an amortised analysis for standard

first-order rewrite system, that is, the only restrictions required are the standard restrictions that (i) the left-hand side of a rule must not be a variable and (ii) no extra variables are introduced in rules.

- The analysis is lifted to relative rewriting, that is, the runtime complexity of a relative TRS \mathcal{R}/\mathcal{S} is measured by the number of rule applications from \mathcal{R} , only. This extension is mainly of practical relevance, as required to obtain an automation of significant strength.
- Finally, the analysis has been implemented and integrated into TCT . We have assessed the viability of the method in context of the TPDB as well as on an independent benchmark.

This paper is structured as follows. In the next section we cover basics. In Sect. 3 we introduce the inference system and prove soundness of the method. In Sect. 4 we detail the implementation of the method and remark on challenges posed by automation. Section 5 provides the experimental assessment of the method. Finally we conclude in Sect. 6, where we also sketch future work. The formal proofs and the full definitions of additional examples have been omitted due to space restrictions. Full details can be found in the second author's master thesis, cf. [13].

2 Preliminaries

We assume familiarity with term rewriting [14, 15] but briefly review basic concepts and notations.

Let \mathcal{V} denote a countably infinite set of variables and \mathcal{F} a signature, such that \mathcal{F} contains at least one constant. The set of terms over \mathcal{F} and \mathcal{V} is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We write $\mathcal{V}\text{ar}(t)$ to denote the set of variables occurring in term t . The *size* $|t|$ of a term is defined as the number of symbols in t .

We suppose $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$, where \mathcal{C} denotes a finite, non-empty set of *constructor symbols*, \mathcal{D} is a finite set of *defined function symbols*, and \uplus denotes disjoint union. Defined function symbols are sometimes referred to as *operators*. A term t is *linear* if every variable in t occurs only once. A term t' is the *linearisation* of a non-linear term t if the variables in t are renamed apart such that t' becomes linear. The notion generalises to sequences of terms. A term $t = f(t_1, \dots, t_k)$ is called *basic*, if f is defined, and all $t_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$. We write $\text{dom}(\sigma)$ ($\text{rg}(\sigma)$) to denote the domain (range) of σ .

Let $\rightarrow \subseteq S \times S$ be a binary relation. We denote by \rightarrow^+ the transitive and by \rightarrow^* the transitive and reflexive closure of \rightarrow . By \rightarrow^n we denote the n -fold application of \rightarrow . If t is in normal form with respect to \rightarrow , we write $s \rightarrow^! t$. We say that \rightarrow is *well-founded* or *terminating* if there is no infinite sequence $s_0 \rightarrow s_1 \rightarrow \dots$. It is *finitely branching* if the set $\{t \mid s \rightarrow t\}$ is finite for each $s \in S$. For two binary relations \rightarrow_A and \rightarrow_B , the relation of \rightarrow_A *relative* to \rightarrow_B is defined by $\rightarrow_A / \rightarrow_B := \rightarrow_B^* \cdot \rightarrow_A \cdot \rightarrow_B^*$.

A *rewrite rule* is a pair $l \rightarrow r$ of terms, such that (i) the root symbol of l is defined, and (ii) $\mathcal{V}\text{ar}(l) \supseteq \mathcal{V}\text{ar}(r)$. A *term rewrite system* (TRS) over \mathcal{F} is a finite set of rewrite rules. Observe that TRSs need not be constructor systems, that is,

arguments of left-hand sides of rules may contain defined symbols. Such function symbols are called *constructor-like*, as below they will be sometimes subject to similar restrictions as constructor symbols.

The set of normal forms of a TRS \mathcal{R} is denoted as $\text{NF}(\mathcal{R})$, or NF for short. We call a substitution σ *normalised with respect to \mathcal{R}* if all terms in the range of σ are ground normal forms of \mathcal{R} . Typically \mathcal{R} is clear from context, so we simply speak of a *normalised* substitution. In the sequel we are concerned with *innermost* rewriting, that is, an eager evaluation strategy. Furthermore, we consider relative rewriting.

A TRS is *left-linear* if all rules are left-linear, it is *non-overlapping* if there is no critical pairs, that is, no ambiguity exists in applying rules. A TRS is *orthogonal* if it is left-linear and non-overlapping. A TRS is *completely defined* if all ground normal-forms are values. Note that an orthogonal TRS is confluent. A TRS is *constructor* if all arguments of left-hand sides are basic.

The *innermost rewrite relation* $\overset{i}{\rightarrow}_{\mathcal{R}}$ of a TRS \mathcal{R} is defined on terms as follows: $s \overset{i}{\rightarrow}_{\mathcal{R}} t$ if there exists a rewrite rule $l \rightarrow r \in \mathcal{R}$, a context C , and a substitution σ such that $s = C[l\sigma]$, $t = C[r\sigma]$, and all proper subterms of $l\sigma$ are normal forms of \mathcal{R} . In order to generalise the innermost rewriting relation to relative rewriting, we introduce the slightly technical construction of the *restricted* rewrite relation [16]. The *restricted rewrite relation* $\overset{Q}{\rightarrow}_{\mathcal{R}}$ is the restriction of $\rightarrow_{\mathcal{R}}$ where all arguments of the redex are in normal form with respect to the TRS \mathcal{Q} . We define the *innermost rewrite relation*, dubbed $\overset{i}{\rightarrow}_{\mathcal{R}/\mathcal{S}}$, of a relative TRS \mathcal{R}/\mathcal{S} as follows.

$$\overset{i}{\rightarrow}_{\mathcal{R}/\mathcal{S}} := \overset{\mathcal{R} \cup \mathcal{S}}{\rightarrow}_{\mathcal{S}}^* \cdot \overset{\mathcal{R} \cup \mathcal{S}}{\rightarrow}_{\mathcal{R}} \cdot \overset{\mathcal{R} \cup \mathcal{S}}{\rightarrow}_{\mathcal{S}}^* .$$

Observe that $\overset{i}{\rightarrow}_{\mathcal{R}} = \overset{i}{\rightarrow}_{\mathcal{R}/\emptyset}$ holds.

Let s and t be terms, such that t is in normal-form. Then a *derivation* $D: s \rightarrow_{\mathcal{R}}^* t$ with respect to a TRS \mathcal{R} is a finite sequence of rewrite steps. The *derivation height* of a term s with respect to a well-founded, finitely branching relation \rightarrow is defined as $\text{dh}(s, \rightarrow) = \max\{n \mid \exists t \ s \rightarrow^n t\}$.

Definition 1. We define the innermost runtime complexity (with respect to \mathcal{R}/\mathcal{S}): $\text{rc}_{\mathcal{R}}(n) := \max\{\text{dh}(t, \overset{i}{\rightarrow}_{\mathcal{R}/\mathcal{S}}) \mid t \text{ is basic and } |t| \leq n\}$.

Intuitively the innermost runtime complexity wrt. \mathcal{R}/\mathcal{S} counts the maximal number of eager evaluation steps in \mathcal{R} in a derivation over $\mathcal{R} \cup \mathcal{S}$. In the definition, we tacitly assume that $\overset{i}{\rightarrow}_{\mathcal{R}/\mathcal{S}}$ is terminating and finitely branching.

For the rest of the paper the relative TRS \mathcal{R}/\mathcal{S} and its signature \mathcal{F} are fixed. In the sequel of the paper, substitutions are assumed to be normalised with respect to $\mathcal{R} \cup \mathcal{S}$.

3 Resource Annotations

In this section, we establish a novel amortised resource analysis for TRSs. This analysis is based on the potential method and coached in an inference system. Firstly, we annotate the (untyped) signature by the prospective resource usage

(Definition 2). Secondly, we define a suitable inference system, akin to a type system. Based on this inference system we delineate a class of *resource bounded* TRSs (Definition 10) for which we deduce polynomial bounds on the innermost runtime complexity for a suitably chosen class of annotations, cf. Theorem 16.

A *resource annotation* \mathbf{p} is a vector $\mathbf{p} = (p_1, \dots, p_k)$ over non-negative rational numbers. The vector \mathbf{p} is also simply called *annotation*. Resource annotations are denoted by $\mathbf{p}, \mathbf{q}, \mathbf{u}, \mathbf{v}, \dots$, possibly extended by subscripts and we write \mathcal{A} for the set of such annotations. For resource annotations (p) of length 1 we write p . We will see that a resource annotation does not change its meaning if zeroes are appended at the end, so, conceptually, we can identify $()$ with (0) and also with 0 . If $\mathbf{p} = (p_1, \dots, p_k)$ we set $k := |\mathbf{p}|$ and $\max \mathbf{p} := \max\{p_i \mid i = 1, \dots, k\}$. We define the notations $\mathbf{p} \leq \mathbf{q}$ and $\mathbf{p} + \mathbf{q}$ and $\lambda \mathbf{p}$ for $\lambda \geq 0$ component-wise, filling up with 0s if needed. So, for example $(1, 2) \leq (1, 2, 3)$ and $(1, 2) + (3, 4, 5) = (4, 6, 5)$.

Definition 2. Let f be a function symbol of arity n . We annotate the arguments and results of f by resource annotations. A (resource) annotation for f , decorated with $k \in \mathbb{Q}^+$, is denoted as $[\mathbf{p}_1 \times \dots \times \mathbf{p}_n] \xrightarrow{k} \mathbf{q}$. The set of annotations is denoted \mathcal{F}_{pol} .

We lift signatures \mathcal{F} to *annotated signatures* $\mathcal{F}: \mathcal{C} \cup \mathcal{D} \rightarrow (\mathcal{P}(\mathcal{F}_{\text{pol}}) \setminus \emptyset)$ by mapping a function symbol to a non-empty set of resource annotations. Hence for any function symbol we allow multiple types. In the context of operators this is also referred to as *resource polymorphism*. The inference system, presented below, mimics a type system, where the provided annotations play the role of types. If the annotation of a constructor or constructor-like symbol f results in \mathbf{q} , there must only be exactly one declaration of the form $[\mathbf{p}_1 \times \dots \times \mathbf{p}_n] \xrightarrow{k} \mathbf{q}$ in $\mathcal{F}(f)$, that is, the annotation has to be *unique*. Moreover, annotations for constructor and constructor-like symbols f must satisfy the *superposition principle*: If f admits the annotations $[\mathbf{p}_1 \times \dots \times \mathbf{p}_n] \xrightarrow{k} \mathbf{q}$ and $[\mathbf{p}'_1 \times \dots \times \mathbf{p}'_n] \xrightarrow{k'} \mathbf{q}'$ then it also has the annotations $[\lambda \mathbf{p}_1 \times \dots \times \lambda \mathbf{p}_n] \xrightarrow{\lambda k} \lambda \mathbf{q}$ ($\lambda \in \mathbb{Q}^+$, $\lambda \geq 0$) and $[\mathbf{p}_1 + \mathbf{p}'_1 \times \dots \times \mathbf{p}_n + \mathbf{p}'_n] \xrightarrow{k+k'} \mathbf{q} + \mathbf{q}'$.

Example 3. Consider the sets $\mathcal{D} = \{\text{enq}, \text{rev}, \text{rev}', \text{snoc}, \text{chk}, \text{hd}, \text{tl}\}$ and $\mathcal{C} = \{\text{nil}, \#, \text{que}, 0, \text{s}\}$, which together make up the signature \mathcal{F} of the motivating example \mathcal{R}_1 in Fig. 1. Annotations of the constructors nil and $\#$ would for example be as follows. $\mathcal{F}(\text{nil}) = \{[\square] \xrightarrow{0} k \mid k \geq 0\}$ and $\mathcal{F}(\#) = \{[0 \times k] \xrightarrow{k} k \mid k \geq 0\}$. These annotations are unique and fulfill the superposition principle. \square

Note that, in view of superposition and uniqueness, the annotations of a given constructor or constructor-like symbol are uniquely determined once we fix the resource annotations for result annotations of the form $(0, \dots, 0, 1)$ (remember the implicit filling up with 0s). An annotated signature \mathcal{F} is simply called *signature*, where we sometimes write $f: [\mathbf{p}_1 \times \dots \times \mathbf{p}_n] \xrightarrow{k} \mathbf{q}$ instead of $[\mathbf{p}_1 \times \dots \times \mathbf{p}_n] \xrightarrow{k} \mathbf{q} \in \mathcal{F}(f)$.

$$\begin{array}{c}
 \frac{f \in \mathcal{C} \cup \mathcal{D} \quad [\mathbf{p}_1 \times \cdots \times \mathbf{p}_n] \xrightarrow{k} \mathbf{q} \in \mathcal{F}(f)}{x_1 : \mathbf{p}_1, \dots, x_n : \mathbf{p}_n \mid^k f(x_1, \dots, x_n) : \mathbf{q}} \text{ (app)} \qquad \frac{\Gamma \mid^k t : \mathbf{q} \quad k' \geq k}{\Gamma \mid^{k'} t : \mathbf{q}} \text{ (w}_1) \\
 \\
 \frac{\text{all } x_i \text{ are fresh} \qquad k = \sum_{i=0}^n k_i \qquad x_1 : \mathbf{p}_1, \dots, x_n : \mathbf{p}_n \mid^{k_0} f(x_1, \dots, x_n) : \mathbf{q} \quad \Gamma_1 \mid^{k_1} t_1 : \mathbf{p}_1 \cdots \Gamma_n \mid^{k_n} t_n : \mathbf{p}_n}{\Gamma_1, \dots, \Gamma_n \mid^k f(t_1, \dots, t_n) : \mathbf{q}} \text{ (comp)} \\
 \\
 \frac{\Gamma \mid^k t : \mathbf{q}}{\Gamma, x : \mathbf{p} \mid^k t : \mathbf{q}} \text{ (w}_4) \qquad \frac{\Gamma, x : \mathbf{r}, y : \mathbf{s} \mid^k t[x, y] : \mathbf{q} \quad \forall(\mathbf{p} \mid \mathbf{r}, \mathbf{s}) \quad x, y \text{ are fresh}}{\Gamma, z : \mathbf{p} \mid^k t[z, z] : \mathbf{q}} \text{ (share)} \\
 \\
 \frac{\Gamma, x : \mathbf{r} \mid^k t : \mathbf{q} \quad \mathbf{p} \geq \mathbf{r}}{\Gamma, x : \mathbf{p} \mid^k t : \mathbf{q}} \text{ (w}_2) \qquad \frac{}{x : \mathbf{q} \mid^0 x : \mathbf{q}} \text{ (var)} \qquad \frac{\Gamma \mid^k t : \mathbf{s} \quad \mathbf{s} \geq \mathbf{q}}{\Gamma \mid^k t : \mathbf{q}} \text{ (w}_3)
 \end{array}$$

Fig. 2. Inference system for term rewrite systems.

The next definition introduces the notion of the potential of a normal form. For rules $f(l_1, \dots, l_n) \rightarrow r$ in non-constructor TRSs the left-hand side $f(l_1, \dots, l_n)$ need not necessarily be basic terms. However, the arguments l_i are deconstructed in the rule (app) that we will see in Fig. 2. This deconstruction may free potential, which needs to be well-defined. This makes it necessary to treat defined function symbols in l_i similar to constructors in the inference system (see Definition 7).

Definition 4. Let $v = f(v_1, \dots, v_n)$ be a normal form and let \mathbf{q} be a resource annotation. We define the potential of v with respect to \mathbf{q} , written $\Phi(v : \mathbf{q})$ by cases. First suppose v contains only constructors or constructor-like symbols. Then the potential is defined recursively.

$$\Phi(v : \mathbf{q}) := k + \Phi(v_1 : \mathbf{p}_1) + \cdots + \Phi(v_n : \mathbf{p}_n) ,$$

where $[\mathbf{p}_1 \times \cdots \times \mathbf{p}_n] \xrightarrow{k} \mathbf{q} \in \mathcal{F}(f)$. Otherwise, we set $\Phi(v : \mathbf{q}) := 0$.

The sharing relation $\forall(\mathbf{p} \mid \mathbf{p}_1, \mathbf{p}_2)$ holds if $\mathbf{p}_1 + \mathbf{p}_2 = \mathbf{p}$.

Lemma 5. Let v be a normal form. If $\forall(\mathbf{p} \mid \mathbf{p}_1, \mathbf{p}_2)$ then $\Phi(v : \mathbf{p}) = \Phi(v : \mathbf{p}_1) + \Phi(v : \mathbf{p}_2)$. Furthermore, if $\mathbf{p} \leq \mathbf{q}$ then $\Phi(v : \mathbf{p}) \leq \Phi(v : \mathbf{q})$.

A (variable) context is a partial mapping from variables \mathcal{V} to annotations. Contexts are denoted by upper-case Greek letters and depicted as sequences of pairs $x : \mathbf{q}$ of variables and annotations, where $x : \mathbf{q}$ in a variable context means that the resource \mathbf{q} can be distributed over all occurrences of the variable x in the term.

Definition 6. Our potential based amortised analysis is coached in an inference system whose rules are given in Fig. 2. Let t be a term and \mathbf{q} a resource annotation. The inference system derives judgements of the form $\Gamma \mid^k t : \mathbf{q}$, where Γ

is a variable context and $k \in \mathbb{Q}^+$ denotes the amortised costs at least required to evaluate t .

Furthermore, we define a subset of the inference rules, free of weakening rules, dubbed the footprint of the judgement, denoted as $\Gamma \Big|_{\text{fp}}^k t: \mathbf{q}$. For the footprint we only consider the inference rules (*app*), (*comp*), (*share*), and (*var*).

Occasionally we omit the amortised costs from both judgements using the notations $\Gamma \vdash t: \mathbf{q}$ and $\Gamma \Big|_{\text{fp}} t: \mathbf{q}$.

To ease the presentation we have omitted certain conditions, like the pairwise disjointedness of $\Gamma_1, \dots, \Gamma_n$ in the rule (*comp*), that make the inference rules deterministic. However, the implementation (see Sect. 4) is deterministic, which removes redundancy in constraint building and thus improves performance. A substitution is called *consistent with Γ* if for all $x \in \text{dom}(\sigma)$ if $\Gamma \vdash x: \mathbf{q}$, then $\Gamma \vdash x\sigma: \mathbf{q}$. Recall that substitutions are assumed to be normalised. Let Γ be a context and let σ be a substitution consistent with Γ . Then $\Phi(\sigma: \Gamma) := \sum_{x \in \text{dom}(\Gamma)} \Phi(x\sigma: \Gamma(x))$.

Definition 7. Let $f(l_1, \dots, l_n) \rightarrow r$, $n \geq 1$, be a rule in the TRS \mathcal{R}/\mathcal{S} . Further suppose $f: [\mathbf{p}_1 \times \dots \times \mathbf{p}_n] \xrightarrow{k} \mathbf{q}$ is a resource annotation for f and let $V := \{y_1, \dots, y_m\}$ denote the set of variables in the left-hand side of the rule. The potential freed by the rule is a pair consisting of a variable context $y_1: \mathbf{r}_1, \dots, y_m: \mathbf{r}_m$ and an amortised cost ℓ , defined as follows:

- The sequence l'_1, \dots, l'_n is a linearisation of l_1, \dots, l_n . Set $Z := \bigcup_{i=1}^n \text{Var}(l'_i)$ and let $Z = \{z_1, \dots, z_{m'}\}$, where $m' \geq m$.
- There exist annotations $\mathbf{s}_1, \dots, \mathbf{s}_{m'}$ such that for all i there exist costs ℓ_i such that $z_1: \mathbf{s}_1, \dots, z_{m'}: \mathbf{s}_{m'} \Big|_{\text{fp}}^{\ell_i} l'_i: \mathbf{p}_i$.
- Let $y_j \in V$ and let $\{z_{j_1}, \dots, z_{j_o}\} \subseteq Z$ be all renamings of y_j . Define annotations $\mathbf{r}_j := \mathbf{s}_{j_1} + \dots + \mathbf{s}_{j_o}$.
- Finally, $\ell := \sum_{i=1}^n \ell_i$.

Example 8. Consider the rule $\text{enq}(\mathbf{s}(n)) \rightarrow \text{snoc}(\text{enq}(n), n)$ in the running example, together with the annotated signature $\text{enq}: [15] \xrightarrow{12} 7$. The left-hand side contains the subterm $\mathbf{s}(n)$. Using the generic annotation $\mathbf{s}: [k] \xrightarrow{k} k$, the footprint $n: k \Big|_{\text{fp}}^k \mathbf{s}(n): k$ is derivable for any $k \geq 0$. Thus, in particular the rule frees the context $n: 15$ and cost 15. \square

Lemma 9. Let $f(l_1, \dots, l_n) \rightarrow r \in \mathcal{R}/\mathcal{S}$ and let $c: [\mathbf{p}_1 \times \dots \times \mathbf{p}_n] \xrightarrow{0} \mathbf{q}$ denote a fresh, cost-free constructor. Let $y_1: \mathbf{r}_1, \dots, y_m: \mathbf{r}_m$ and ℓ be freed by the rule. We obtain: $y_1: \mathbf{r}_1, \dots, y_m: \mathbf{r}_m \Big|_{\text{fp}}^{\ell} c(l_1, \dots, l_n): \mathbf{q}$.

Based on Definition 7 we can now succinctly define resource boundedness of a TRS. The definition constitutes a non-trivial generalisation of Definition 11 in [8]. First the input TRS need no longer be sorted. Second the restriction on constructor TRSs has been dropped and finally, the definition has been extended to handle relative rewriting.

Definition 10. Let \mathcal{R}/\mathcal{S} be a relative TRS, let \mathcal{F} be a signature and let $f \in \mathcal{F}$. An annotation $[\mathbf{p}_1 \times \cdots \times \mathbf{p}_n] \xrightarrow{k} \mathbf{q} \in \mathcal{F}(f)$ is called *resource bounded* if for any rule $f(l_1, \dots, l_n) \rightarrow r \in \mathcal{R} \cup \mathcal{S}$, we have

$$y_1: \mathbf{r}_1, \dots, y_l: \mathbf{r}_l \mid \frac{k + \ell - K^{\text{rule}}}{r: \mathbf{q}},$$

where $y_1: \mathbf{r}_1, \dots, y_l: \mathbf{r}_l$ and ℓ are freed by the rule if $n \geq 1$ and $\ell = 0$ otherwise. Here, the cost K^{rule} for the application of the rule is defined as follows: (i) $K^{\text{rule}} := 1$ iff $f(l_1, \dots, l_n) \rightarrow r \in \mathcal{R}$ and (ii) $K^{\text{rule}} := 0$ iff $f(l_1, \dots, l_n) \rightarrow r \in \mathcal{S}$. We call an annotation *cost-free resource bounded* if the cost K^{rule} is always set to zero.

A function symbol f is called (*cost-free*) *resource bounded* if any resource annotation in $\mathcal{F}(f)$ is (cost-free) resource bounded. Finally, \mathcal{R}/\mathcal{S} is called *resource bounded*, or simply *bounded* if any $f \in \mathcal{F}$ is resource bounded. Observe that boundedness of \mathcal{R}/\mathcal{S} entails that the application of rules in the strict part \mathcal{R} is counted, while the weak part \mathcal{S} is not counted.

In a nutshell, the method works as follows: Suppose the judgement $\Gamma \mid^{k'} t: \mathbf{q}$ is derivable and suppose σ is consistent with Γ . The constant k' is an upper-bound to the amortised cost required for reducing t to normal form. Below we will prove that the derivation height of $t\sigma$ (with respect to innermost rewriting) is bounded by the difference in the potential before and after the evaluation plus k' . Thus if the sum of the potentials of the arguments of $t\sigma$ is in $\mathcal{O}(n^k)$, where n is the size of the arguments and k the maximal length of the resource annotations needed, then the innermost runtime complexity of \mathcal{R}/\mathcal{S} lies in $\mathcal{O}(n^k)$.

More precisely consider the `comp` rule. First note that this rule is only applicable if $f(t_1, \dots, t_n)$ is linear, which can always be obtained by the use of the sharing rule. Now the rule embodies that the amortised costs k' required to evaluate $t\sigma$ can be split into those costs k'_i ($i \geq 1$) required for the normalisation of the arguments and the cost k'_0 of the evaluation of the operator f . Furthermore the potential provided in the context $\Gamma_1, \dots, \Gamma_n$ is suitably distributed. Finally the potential which remains after the evaluation of the arguments is made available for the evaluation of the operator f .

Before we proceed with the formal proof of this intuition, we exemplify the method on the running example.

Example 11 (continued from Example 3). $\top_{\mathcal{CT}}$ derives the following annotations for the operators in the running example.

$$\begin{array}{lll} \text{enq}: [15] \xrightarrow{12} 7 & \text{rev}: [1] \xrightarrow{4} 0 & \text{rev}': [1 \times 0] \xrightarrow{2} 0 \\ \text{snoc}: [7 \times 0] \xrightarrow{14} 7 & \text{hd}: [11] \xrightarrow{9} 0 & \text{tl}: [11] \xrightarrow{3} 1 \quad \square \end{array}$$

We consider resource boundedness of \mathcal{R}_1 with respect to the given (monomorphic) annotated signatures of Example 11. For simplicity we restrict to boundedness of `enq`. We leave it to the reader to check the other cases. In addition to

the annotations for constructor symbols (cf. Example 3) we can always assume the presence of zero-cost annotations, e.g. $\sharp: [0 \times 0] \xrightarrow{0} 0$. Observe that Rule 6 frees the context $n:15$ and cost 15. Thus, we obtain the following derivation.

$$\frac{\frac{\text{snoc}: [7 \times 0] \xrightarrow{14} 7}{q: 7, m: 0 \mid^{14} \text{snoc}(q, m): 7} \quad (\text{app}) \quad \frac{}{n_2: 0 \mid^0 n_2: 0} \quad (\text{var}) \quad \frac{\text{enq}: [15] \xrightarrow{12} 7}{n_1: 15 \mid^{12} \text{enq}(n_1): 7} \quad (\text{app})}{\frac{n_1: 15, n_2: 0 \mid^{26} \text{snoc}(\text{enq}(n_1), n_2): 7}{n: 15 \mid^{26} \text{snoc}(\text{enq}(n), n): 7} \quad (\text{share})} \quad (\text{comp})$$

In comparison to [8, Example 13], where the annotations were found manually, we note that the use of the interleaving operation [8] has been avoided. This is due to the more general class of annotations considered in our prototype implementation (see Sect. 4).

The footprint relation forms a restriction of the judgement \vdash without the use of weakening. Hence the footprint allows a precise control of the resources stored in the substitutions, as indicated by the next lemma.

Lemma 12. *Let t be a normal form w.r.t. \mathcal{R} , where t consists of constructor or constructor-like symbols only. If $\Gamma \mid_{\text{fp}}^k t: \mathbf{q}$, then $\Phi(t\sigma: \mathbf{q}) = \Phi(\sigma: \Gamma) + k$.*

We state the following substitution lemma. The lemma follows by simple induction on t .

Lemma 13. *Let Γ be a context and let σ be a substitution consistent with Γ . Then $\Gamma \mid t: \mathbf{q}$ implies $\vdash t\sigma: \mathbf{q}$.*

We establish soundness with respect to relative innermost rewriting.

Theorem 14. *Let \mathcal{R}/\mathcal{S} be a resource bounded TRS and let σ be a normalised such that σ is consistent with the context Γ . Suppose $\Gamma \mid^k t: \mathbf{p}$ and $t\sigma \xrightarrow{i}_{\mathcal{R}}^K u\tau$, $K \in \{0, 1\}$ for a normalising substitution τ . Then there exists a context Δ such that $\Delta \mid^\ell t: \mathbf{q}$ is derivable and $\Phi(\sigma: \Gamma) + k - \Phi(\tau: \Delta) - \ell \geq K$.*

Proof. Let Π denote the derivation of the judgement $\Gamma \mid^k t: \mathbf{q}$. The proof proceeds by case distinction on derivation $D: t\sigma \xrightarrow{i}_{\mathcal{R}}^K u\tau$ and side-induction on Π . The proof proceeds by case distinction on D and induction on the length of Π . \square

The next corollary is an immediate consequence of the theorem, highlighting the connection to similar soundness results in the literature.

Corollary 15. *Let \mathcal{R}/\mathcal{S} be a bounded TRS and let σ be a normalising substitution consistent with the context Γ . Suppose $\Gamma \mid^k t: \mathbf{q}$ and $D: t\sigma \xrightarrow{i}_{\mathcal{R}/\mathcal{S}}^1 v \in \text{NF}$. Then (i) $\vdash v: \mathbf{q}$ and (ii) $\Phi(\sigma: \Gamma) - \Phi(v: \mathbf{q}) + k \geq |D|$ hold. \square*

The next theorem defines suitable constraints on the resource annotations to deduce polynomial innermost runtime from Theorem 14. Its proof follows the pattern of the proof of Theorem 14 in [8].

Theorem 16. *Suppose that for each constructor c with $[\mathbf{p}_1 \times \dots \times \mathbf{p}_n] \xrightarrow{k'} \mathbf{q} \in \mathcal{F}(c)$, there exists $\mathbf{r}_i \in \mathcal{A}$ such that $\mathbf{p}_i \leq \mathbf{q} + \mathbf{r}_i$ where $\max \mathbf{r}_i \leq \max \mathbf{q} =: r$ and $p \leq r$ with $|\mathbf{r}_i| < |\mathbf{q}| =: k$. Then $\Phi(v; \mathbf{q}) \leq r|v|^k$, and thus the innermost runtime complexity of the TRS under investigation is in $\mathcal{O}(n^k)$.*

Proof. The theorem follows the pattern of the proof of Theorem 14 in [8]. \square

We note that our running example satisfies the premise of Theorem 16. Thus the linear bound on the innermost runtime complexity of the running example \mathcal{R}_1 follows. The next example clarifies that without further assumptions potentials are not restricted to polynomials.

Example 17. Consider that we annotate the constructors for natural numbers as $0: [] \xrightarrow{0} \mathbf{p}$ and $s: [2\mathbf{p}] \xrightarrow{p_1} \mathbf{p}$, where $\mathbf{p} = (p_1, \dots, p_k)$. We then have, for example, $\Phi(t; 1) = 2^v - 1$, where v is the value represented by t . \square

4 Implementation

In this section we describe the details of important implementation issues. The realisations of the presented method can be seen twofold. On one hand we have a standalone program which tries to directly annotate the given TRS. While on the other hand the integration into \mathcal{TCT} [11] uses relative rewriting. Clearly, as an integration into \mathcal{TCT} was planned from the beginning, the language used for the implementation of the amortised resource analysis module is Haskell¹. The modular design of \mathcal{TCT} eased the integration tremendously.

The central idea of the implementation is the collection of all signatures and arising constraints occurring in the inference tree derivations. To guarantee resource boundedness further constraints are added such that uniqueness and superposition of constructors (cf. Sect. 3) is demanded and polynomial bounds on the runtime complexity are guaranteed (cf. Theorem 16).

Inference Tree Derivation and Resource Boundedness. To be able to apply the inference rules the expected root judgement of each rule is generated (as in Example 11) by the program and the inference rules of Fig. 2 are applied. To gain determinism the inference rules are ordered in the following way. The share-rule has highest priority, followed by `app`, `var`, `comp` and `w4`. In each step the first applicable rule is used while the remaining weakening rules `w1`, `w2` and `w3` are integrated in the aforementioned ones. For each application of an inference rule the emerging constraints are collected.

To ensure monomorphic typing of function signatures we keep track of a list of signatures. It uses variables in lieu of actual vectors. For each signature occurrence of defined function symbols the system refers to the corresponding entry in the list of signatures. Therefore, for each defined function symbol only one signature is added to the list of signatures. If the function occurs multiple

¹ See <http://haskell.org/>.

times, the same references are used. Unlike defined function symbols multiple signature declarations of constructors are allowed, and thus each occurrence adds one signature to the list.

For the integration into TCT we utilise the relative rewriting formulation. Instead of requiring all strict rules to be resource bounded, we weaken this requirement to have at least one strict rule being actually resource bounded, while the other rules may be annotated cost-free resource bounded. The SMT solver chooses which rule will be resource bounded. Clearly, this eases the constraint problem which is given to the SMT solver.

Superposition of Constructors. Recall that constructor and constructor-like symbols f must satisfy the superposition principle. Therefore, for each annotation $[\mathbf{p}_1 \times \dots \times \mathbf{p}_n] \xrightarrow{k} \mathbf{q}$ of f it must be ensured that there is no annotation $[\lambda \cdot \mathbf{p}_1 \times \dots \times \lambda \cdot \mathbf{p}_n] \xrightarrow{\lambda \cdot k} \mathbf{q}'$ with $\lambda \in \mathbb{Q}^+$ and $\mathbf{q} \neq \lambda \cdot \mathbf{q}'$ in the corresponding set of annotated signatures. Therefore, for every pair $(\mathbf{q}, \mathbf{q}')$ with $\mathbf{q}' \geq \mathbf{q}$ and $\mathbf{q} > 0$ either for every $\lambda > 0$: $\mathbf{q}' \neq \lambda \cdot \mathbf{q}$ or if $\mathbf{q}' = \lambda \cdot \mathbf{q}$ then the annotation must be of the form $[\lambda \cdot \mathbf{p}_1 \times \dots \times \lambda \cdot \mathbf{p}_n] \xrightarrow{\lambda \cdot k} \lambda \cdot \mathbf{q}$.

A naive approach is adding corresponding constraints for every pair of return annotations of a constructor symbol. This leads to universal quantifiers due to the scalar multiplication, which however, are available as binders in modern SMT solvers [17]. Early experiments revealed their bad performance. Overcoming this issue using Farkas' Lemma [18] is not possible here. Thus, we developed the heuristic of spanning up a vector space using unit vectors for the annotation of the return types for each constructor. Each annotated signature of such a symbol must be a linear combination of the base signatures.

Both methods, universal quantifiers and base signatures lead to non-linear constraint problems. However, these can be handled by some SMT solvers². Thus, in contrast to the techniques presented in [3, 19, 20], which restrict the potential function to pre-determined data structures, like lists or binary trees, our method allows any kind of data structure to be annotated.

Example 18. Consider the base constructors $\#_1: [(0, 0) \times (1, 0)] \xrightarrow{1} (1, 0)$ and $\#_2: [(0, 0) \times (2, 1)] \xrightarrow{1} (0, 1)$ for a constructor $\#$. An actual instance of an annotated signature is $n_1 \cdot \#_1 + n_2 \cdot \#_2$ with $n_1, n_2 \in \mathbb{N}$. As the return types can be seen as unit vectors of a Cartesian coordinate system, the superposition and uniqueness properties hold. \square

Cost-Free Function Symbols. Inspired by Hoffmann [19, p. 93ff] we additionally implemented a cost-free inference tree derivation when searching for non-linear bounds. The idea is that for many non-tail recursive functions the freed potential must be the one of the original function call plus the potential that gets passed on.

² We use the SMT Solvers z3 (<https://github.com/Z3Prover/z3/wiki>) and MiniSmt (<http://cl-informatik.uibk.ac.at/software/minismt/>).

$$\frac{
 \begin{array}{l}
 [\mathbf{p}_1^{cf} \times \dots \times \mathbf{p}_n^{cf}] \xrightarrow{k^{cf}} \mathbf{q}^{cf} \in \mathcal{F}^{cf}(f) \\
 [\mathbf{p}_1 \times \dots \times \mathbf{p}_n] \xrightarrow{k} \mathbf{q} \in \mathcal{F}(f)
 \end{array}
 \quad
 y_1: \mathbf{r}_1, \dots, y_l: \mathbf{r}_l \mid \xrightarrow{k+\ell} r: \mathbf{q}
 }{
 x_1: \mathbf{p}_1, \dots, x_n: \mathbf{p}_n \mid \xrightarrow{k} f(x_1, \dots, x_n): \mathbf{q}
 }$$

Fig. 3. Additional **app** rule for cost-free derivation, where $f \in \mathcal{C} \cup \mathcal{D}$.

The inference rules are extended by an additional **app**-rule, which separates the function signature into two parts, cf. Fig. 3. On the left there are the monomorphic and cost-free signatures while on the right a cost-free part is added. For every application of the rule the newly generated cost-free signature annotation must be cost-free resource bounded, for this the cost-free type judgement indicated has to be derived for any rule $f(l_1, \dots, l_n) \rightarrow r$ and freed context $y_1: \mathbf{r}_1, \dots, y_l: \mathbf{r}_l$ and cost ℓ . Thus, the new set of annotations for a defined function symbols f is given by the following set, cf. [19, p. 93].

$$\{[\mathbf{p}_1 + \lambda \cdot \mathbf{p}_1^{cf} \times \dots \times \mathbf{p}_n + \lambda \cdot \mathbf{p}_n^{cf}] \xrightarrow{k+\lambda \cdot k^{cf}} \mathbf{q} + \lambda \cdot \mathbf{q}^{cf} \mid \lambda \in \mathbb{Q}^+, \lambda \geq 0\}.$$

The decision of which **app** rule is applied utilises the strongly connected component (SCC) of the call graph analysis as done in [19, p. 93ff].

Alternative Implementation of the Superposition Principle. Similar to [8, 19] we integrated the additive shift $\triangleleft(\mathbf{p})$ and interleaving $\mathbf{p} \parallel \mathbf{q}$ for constructors when type information is given. Here $\triangleleft(p_1, \dots, p_k) := (p_1 + p_2, p_2 + p_3, \dots, p_{k-1} + p_k, p_k)$ and $\mathbf{p} \parallel \mathbf{q} := (p_1, q_1, p_2, q_2, \dots, p_k, q_k)$, where the shorter of the two vectors is padded with 0s. These heuristics are designed such that the superposition principle holds, without the need of base annotations. Therefore, the constraint problem automatically becomes linear whenever these heuristics are used which tremendously reduces the execution times.

However, according to the experiments (see the detail results online) these heuristics are only rarely applicable and often require comprehensive type information. This additional information allows to separate constructors named alike but with different types. For instance, a list of lists can then have different base annotations compared to a simple list, even though the constructors have the same name. The rather poor performance of these heuristics in the presence of only generic type information came as a surprise to us. However, in hindsight it clearly showcases the importance of comprehensive type information (as e.g. demanded by RaML) for the efficiency of automation of resource analysis in functional programming.

5 Experimental Evaluation

In this section we will have a look at how the amortised analysis deals with some selected examples including the paper's running example **queue**. All experiments³

³ Detailed data is available at <http://cl-informatik.uibk.ac.at/software/tct/experiments/ara.flops/>.

Example	$O(1)$	$O(n^1)$	$O(n^2)$	$O(n^3)$	$O(n^{\geq 4})$	Fail
#Systems	2	59	17	21	8	33
Time (in s)	0.05	0.54	2.86	5.06	10.14	58.30

Fig. 4. Experimental evaluation of \mathcal{TCT} with ARA.

were conducted on a machine with an Intel Xeon CPU E5-2630 v3 @ 2.40 GHz (32 threads) and 64 GB RAM. The timeout was set to 60 s. For benchmarking we use the runtime complexity innermost rewriting folder of the TPDB⁴ as well as a collection consisting of 140 TRSs representing first-order functional programs [21, 22], transformations from higher-order programs [23], or RaML programs [20] and interesting examples from the TPDB. We compared the competition version of \mathcal{TCT} 2016 to the current version of \mathcal{TCT} with and without (w/o) the amortised resource analysis (ARA), as well as the output of AProVE as presented in [24]⁵. Figure 4 shows the results of the experiments conducted for the \mathcal{TCT} with ARA. In a companion paper, we have studied *best case* complexity and suitable adapted amortised resource analysis to obtain lower bounds on the best case complexity. Therefore, the standalone tool is also able to infer best case complexity bounds for TRSs [12].

#3.42 – Binary Representation. Given a number n in unary encoding as input, the TRS computes the binary representation $(n)_2$ by repeatedly halving n and computing the last bit, see the Appendix for the TRS. The optimal runtime complexity of \mathcal{R}_1 is linear in n . For this, first observe that the evaluation of $\text{half}(s^m(0))$ and $\text{lastbit}(s^m(0))$ requires about m steps in total. Secondly, n is halved in each iteration and thus the number of steps can be estimated by $\sum_{i=0}^k 2^i$, where $k := |(n)_2|$. As the geometric sum computes to $2 \cdot 2^k - 1$, the claim follows. Such a precise analysis is enabled by an amortised analysis, which takes the sequence of subsequent function calls and their respective arguments into account. Compared to former versions of \mathcal{TCT} which reported $O(n^2)$ we find this optimal linear bound of $O(n)$ when ARA is enabled. Furthermore, the best case analysis of ARA shows that this bound is tight by returning $\Omega(n)$. Similarly AProVE [25] yields the tight bound employing a size abstraction to *integer transition systems* (ITSs for short), cf. [24]. The resulting ITSs are then solved with CoFloCo [26], which also embodies an amortisation analysis.

bfs.raml – Depth/Breadth-First Search. This TRS is a translation of depth-first search (DFS) and breadth-first search (BFS) from RaML syntax, see Appendix, and can be found in the TPDB. Note that the TRS uses strict rules for the equality check which recurses on the given data structure. In DFS a binary tree is searched one branch after the other for a matching entry while BFS uses two

⁴ We refer to Version 10.4 of the Termination Problem Database, available from <http://cl2-informatik.uibk.ac.at/mercurial.cgi/TPDB>.

⁵ See https://aprove-developers.github.io/trs_complexity_via_its/ for detailed results of AProVE. Timeout: 300 s, Intel Xeon with 4 cores at 2.33 GHz and 16 GB of RAM.

lists to keep track of nodes of a binary tree to be visited. The first one is used to traverse on the nodes of the current depth, whereas the second list collects all nodes of the next depth to visit. After each iteration the futurelist is reversed. Further, note that BFS is called twice in the function `bfs2`. \mathcal{TCT} with ARA is the only tool which is able to infer a complexity bound of $O(n^3)$.

insertionsort.raml/splitandsort.raml – Sorting. Insertionsort has quadratic runtime complexity, although \mathcal{TCT} with ARA using the default setup can only find a cubic upper bound, as it handles the trade off between execution time and tightness of the bound. If \mathcal{TCT} is triggered to find the best bound within the timeout, it will infer $O(n^2)$ as AProVE does. This bound is tight [19, p. 158ff]. The best case analysis finds a linear lower bound for this implementation of insertionsort. `splitandsort.raml` first groups the input by a specified key and then sorts each grouped list using quicksort. The optimal runtime complexity for this program is $O(n^2)$ [19, 158ff]. Although far from being optimal, \mathcal{TCT} with ARA is able to find the worst case upper bound $O(n^5)$, whereas AProVE infers a cubic bound.

tpa2 – Multiple Subtraction. This TRS from the TPDB iterates subtraction until no more rules can be applied. The latest version of \mathcal{TCT} with ARA is in comparison to an older version able to solve the problem. The inferred quadratic worst case bound coincides with the bounds provided by AProVE.

matrix.raml – Matrix Operations. This TRS implements transposing of matrices and matrix multiplications for a list of matrices, three matrices and two matrices, see the Appendix for an excerpt in RaML syntax of the implemented matrix multiplication for two matrices, of which the second one is already transposed. The program maps over the matrix `m1` line by line, for each line mapping over matrix `m2` calling `mult` on the corresponding entries. Clearly, if the `*`-function is seen as one operation, as in the TRS, this program has cubic worst case runtime complexity. Due to ARA, the latest version of \mathcal{TCT} can now handle this TRS and returns a complexity bound of $O(n^7)$ in the default setup, but when the best bound is looked for, \mathcal{TCT} returns the asymptotically optimal upper bound defined by the list matrix multiplication of $O(n^4)$. Neither the older version of \mathcal{TCT} nor AProVE is able to find any upper bound for this TRS.

Experimental Evaluation. We have conducted several further experiments on the TPDB, as well as on the smaller testbed composed of interesting examples with the focus on program translations. Over the last year the strategy of \mathcal{TCT} was adapted to focus on TRSs which were translated from functional programs. Thus, the examples which can be solved are distinct from the \mathcal{TCT} competition strategy of 2016 to a great extent. Due to ARA the latest competition strategy of \mathcal{TCT} can solve 5 more examples of the TPDB than without ARA and for 14 examples a better bound can be inferred. On the small testbed \mathcal{TCT} with ARA can find better bounds for 22 examples in contrast to \mathcal{TCT} without ARA and additionally `bfs.raml` can be solved. For further experiments see the detailed results.

6 Conclusion

In this paper we have established a novel automated amortised cost analysis for term rewriting. In doing so we have not only implemented the methods detailed in earlier work [8], but also generalised the theoretical basis considerably. We have provided a prototype implementation and integrated into \mathcal{TCT} .

More precisely, we have extended the method of amortised resource analysis to *unrestricted* term rewrite systems, thus overcoming typical restrictions of functional programs like left-linearity, pattern based, non-ambiguity, etc. This extension is non-trivial and generalises earlier results in the literature. Furthermore, we have lifted the method to relative rewriting. The latter is the prerequisite to a *modular* resource analysis, which we have provided through the integration into \mathcal{TCT} . The provided integration of amortised resource analysis into \mathcal{TCT} has led to an increase in overall strength of the tool (in comparison to the latest version without ARA and the current version of AProVE). Furthermore in a significant amount of cases we could find better bounds than before.

In future work we want to focus on lifting the provided amortised analysis in two ways. First we want to extend the provided univariate analysis to a multivariate analysis akin the analysis provided in RaML. The theoretical foundation for this has already been provided by Hofmann and Moser [9]. However efficient automation of the method proposed in [9] requires some sophistication. Secondly, we aim to overcome the restriction to constant amortised analysis and provide an automated (or at least automatable) method establishing logarithmic amortised analysis. This aims at closing the significant gap of existing methods in contrast to the origin of amortised analysis [1, 2], compare also [27].

References

1. Sleator, D., Tarjan, R.: Self-adjusting binary trees. In: Proceedings of the 15th STOC, pp. 235–245. ACM (1983)
2. Tarjan, R.: Amortized computational complexity. *SIAM J. Alg. Disc. Methods* **6**(2), 306–318 (1985)
3. Hoffmann, J., Das, A., Weng, S.: Towards automatic resource bound analysis for OCaml. In: Proceedings of the 44th POPL, pp. 359–373. ACM (2017)
4. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-form upper bounds in static cost analysis. *JAR* **46**(2), 161–203 (2011)
5. Flores-Montoya, A.: Upper and lower amortized cost bounds of programs expressed as cost relations. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 254–273. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_16
6. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: Proceedings of the Software Engineering. LNI, vol. 252, pp. 101–102 (2016)
7. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1999)
8. Hofmann, M., Moser, G.: Amortised resource analysis and typed polynomial interpretations. In: Dowek, G. (ed.) RTA 2014. LNCS, vol. 8560, pp. 272–286. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_19

9. Hofmann, M., Moser, G.: Multivariate amortised resource analysis for term rewrite systems. In: Proceedings of the 13th TLCA. LIPIcs, vol. 38, pp. 241–256 (2015)
10. Avanzini, M., Lago, U.D.: Automating sized-type inference for complexity analysis. In: PACMPL, vol. 1, pp. 43:1–43:29 (2017)
11. Avanzini, M., Moser, G., Schaper, M.: TcT: tyrolean complexity tool. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 407–423. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_24
12. Moser, G., Schneckenreither, M.: Amortised analysis for bestcase lowerbounds (2018, submitted)
13. Schneckenreither, M.: Amortized resource analysis for term rewrite systems. Master’s thesis, University of Innsbruck (2018). https://www.uibk.ac.at/wipl/team/team/docs/masterthesis_schneckenreither.pdf
14. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
15. TeReSe: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press (2003)
16. Thiemann, R.: The DP framework for proving termination of term rewriting. Ph.D. thesis, University of Aachen, Department of Computer Science (2007)
17. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
18. Farkas, J.: Theorie der einfachen ungleichungen. Journal für die reine und angewandte Mathematik **124**, 1–27 (1902)
19. Hoffmann, J.: Types with potential: polynomial resource bounds via automatic amortized analysis. Ph.D. thesis, Ludwig-Maximilians-Universität München (2011)
20. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. TOPLAS **34**(3), 14 (2012)
21. Glenstrup, A.: Terminator II: stopping partial evaluation of fully recursive programs. Master’s thesis, Technical report DIKU-TR-99/8, DIKU (1999)
22. Frederiksen, C.: Automatic runtime analysis for first order functional programs. Master’s thesis, DIKU TOPPS D-470, DIKU (2002)
23. Avanzini, M., Lago, U.D., Moser, G.: Analysing the complexity of functional programs: higher-order meets first-order. In: Proceedings of the 20th ICFP, pp. 152–164. ACM (2015)
24. Naaf, M., Frohn, F., Brockschmidt, M., Fuhs, C., Giesl, J.: Complexity analysis for term rewriting by integer transition systems. In: Dixon, C., Finger, M. (eds.) FroCoS 2017. LNCS (LNAI), vol. 10483, pp. 132–150. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66167-4_8
25. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing program termination and complexity automatically with approve. JAR **58**(1), 3–31 (2017)
26. Flores-Montoya, A.: CoFloCo: system description. In: 15th International Workshop on Termination, vol. 20 (2016)
27. Nipkow, T.: Amortized complexity verified. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 310–324. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_21



A Common Framework Using Expected Types for Several Type Debugging Approaches

Kanae Tsushima¹(✉) and Olaf Chitil²

¹ National Institute of Informatics, Tokyo, Japan
k.tsushima@nii.ac.jp

² University of Kent, Canterbury, UK
O.Chitil@kent.ac.uk

Abstract. Many different approaches to type error debugging were developed independently. In this paper, we describe a new common framework for several type error debugging approaches. For this purpose, we introduce expected types from the outer context and propose a method for obtaining them. Using expected types, we develop three type error debugging approaches: enumeration of type error messages, type error slicing and (improved) interactive type error debugging. Based on our idea we implemented prototypes and confirm that the framework works well for type debugging.

1 Introduction

The Hindley-Milner type system is the core of the type systems of many statically typed functional programming languages such as ML, OCaml and Haskell. The programmer does not have to write any type annotations in the program; instead most general types are inferred automatically. Functions defined in the program can be used with many different types.

However, type error debugging is a well-known problem: When a program cannot be typed, the type error messages produced by the compiler often do not help much with locating and fixing the cause(s). Consider the following OCaml program:

```
let rec f lst n = match lst with
| [] -> []
| fst :: rest -> (fst ^ n) :: (f rest n) in
f [2]
```

We assume that the programmer intended to define the function that maps a list of numbers to a list of squared numbers (e.g., `f [3] 2 = [9]`). Hence the programmer's intended type of `f` is $int\ list \rightarrow int \rightarrow int\ list$. However, in OCaml `^` is the string concatenation function. Therefore, the type of the function `f` is

inferred as $string\ list \rightarrow string \rightarrow string\ list$. The OCaml compiler returns the following type error message:

```
f [2]  ;;
Error: This expression has type int but
       an expression was expected of type string
```

The message states that the underlined expression 2 has type *int*, but it is expected to have type *string* because of other parts of this program. The message does not substantially help the programmer: they will wonder why 2 should have type *string*.

The starting point for our work is the principal typing tree that was introduced by Chitil [2] for interactive type error debugging.

A principal typing $\Gamma \vdash \tau$ of some expression e consists of a type τ and an environment Γ which gives types to all free variables of e . All typings for an expression are instances of its principal typing. For example, the principal typing for a variable x is $\{x : \alpha\} \vdash \alpha$, where α is a type variable. In contrast, the better known principal type τ is just the most general type for a given expression e and given environment Γ .

The principal typing tree is the syntax tree of a program where each node describes a subexpression of the program and includes the principal typing for this subexpression. Chitil applied algorithmic debugging [9] to the tree to locate the source of a type error. Tsushima and Asai proposed an approach to construct the principal typing tree using the compiler's type inferencer and implemented a type error debugger for OCaml [12]. The implemented type debugger has been used in classes at Ochanomizu University for the past 5 years by approximately 200 novice programmers [5].

In this paper we claim that **expected typings** are also useful for type debugging. Expected typings are duals to principal typings. Let us consider a program $C[e]$ that consists of an expression e and its context C . The expression e on its own has a principal typing. The expected typing of e is the type of the hole of the context C , disregarding e . In the previous example the principal type of 2 is *int*, but its expected type, according to the context, is *string*. The type error message of the OCaml compiler actually stated both types.

In this paper we introduce the **type debugging information tree**. The tree for our example program is shown in Fig. 1. The tree includes for each subexpression both its principal and expected typing (Sect. 3).

We show how the tree can be used to realise the following type error debugging approaches:

1. Enumeration of type error messages.
Type error messages show two conflicting types. We can obtain both types from the type debugging information tree (Sect. 4.1).
2. Type error slicing.
Only well-typed contexts yield expected typings. Hence program parts that do not yield expected typings do not contribute to a conflict between principal and expected typing (Sect. 4.2).

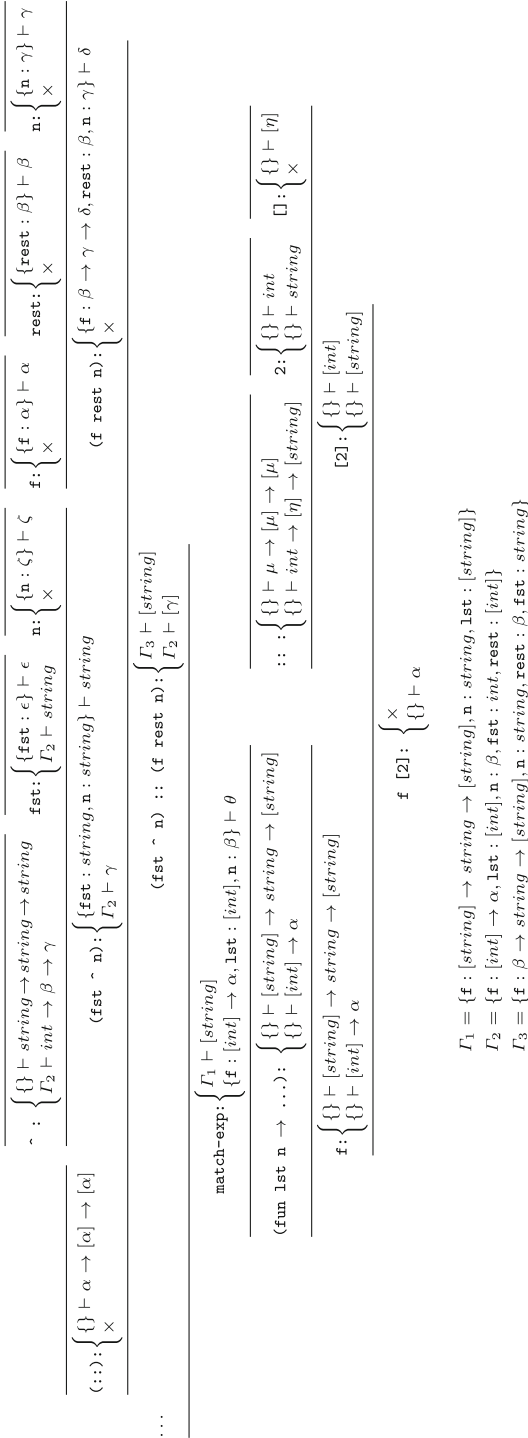


Fig. 1. Type debugging information tree for the program from the Introduction

3. Interactive type error debugging.

We can use the programmer’s intended types to derive expected types. With expected types, we can efficiently narrow the area to debug (Sect. 4.3).

2 Language and Principal Typing Tree (PTT)

We describe our framework for a core functional language, a small subset of OCaml, as defined in Fig. 2. Every expression construct is annotated with some unique location information l . To save space, we use Haskell’s notation for list types (e.g., instead of $int\ list$ we write $[int]$). A program S consists of a sequence of function definitions defined by a recursive let construct. However, this let-rec construct does not appear in any of the trees that we define in the following sections. Instead, we unfold the definition of a let-rec defined variable at every use occurrence in the tree. Note that this ‘unfolding’ is only conceptual to obtain a simple tree structure; the actual implementation handles each let-rec construct only once and actually constructs a directed graph, which, unfolded, yields a tree.

Figure 1 demonstrates this unfolding for the program in the Introduction. It shows part of the type debugging information tree where the tree node for \mathbf{f} has the definition of \mathbf{f} as child node.

$p ::= c$	(constant)
v	(variable)
$p ::= p$	(cons pattern)
(p, p)	(tuple pattern)
$M ::= c^l$	(constant)
v^l	(variable)
$\text{fun}^l x \rightarrow M$	(function)
$(M M)^l$	(application)
$(M, M)^l$	(tuple)
$\text{if}^l M \text{ then } M \text{ else } M$	(if expression)
$\text{match}^l M \text{ with } p \rightarrow M$	(match expression)
$l ::= \text{location number}$	
$S ::= \text{let rec } x = M \text{ in } S$	(let expression)
M	
$\tau ::= \text{int} \mid \text{bool} \mid \dots$	(constant type)
α	(type variable)
$\tau \rightarrow \tau$	(function type)
$\tau * \tau$	(tuple type)
$[\tau]$	(list type)

Fig. 2. OCaml-like functional language

$$\frac{\frac{\frac{\overline{y:\{x: int, y: int\} \vdash int}}{\quad} \quad \frac{\overline{+\{x: int, y: int\} \vdash int \rightarrow int \rightarrow int}}{\quad} \quad \frac{\overline{x:\{x: int, y: int\} \vdash int}}{\quad}}{\frac{y + x:\{x: int, y: int\} \vdash int}{\quad}}}{\frac{(\text{fun } y \rightarrow y + x):\{x: int\} \vdash int \rightarrow int}{\quad}}}{\frac{(\text{fun } x \rightarrow (\text{fun } y \rightarrow y + x)):\{\} \vdash int \rightarrow int \rightarrow int}{\quad}} \quad \frac{\overline{\text{true}:\{\} \vdash bool}}{\quad}}{\frac{(\text{fun } x \rightarrow (\text{fun } y \rightarrow y + x)) \text{ true}:\times}{\quad}}$$

Fig. 3. Standard type inference tree of $(\text{fun } x \rightarrow (\text{fun } y \rightarrow y + x)) \text{ true}$

$$\frac{\frac{\frac{\overline{y:\{y: \beta\} \vdash \beta}}{\quad} \quad \frac{\overline{+\{\} \vdash int \rightarrow int \rightarrow int}}{\quad} \quad \frac{\overline{x:\{x: \alpha\} \vdash \alpha}}{\quad}}{\frac{y + x:\{x: int, y: int\} \vdash int}{\quad}}}{\frac{(\text{fun } y \rightarrow y + x):\{x: int\} \vdash int \rightarrow int}{\quad}}}{\frac{(\text{fun } x \rightarrow (\text{fun } y \rightarrow y + x)):\{\} \vdash int \rightarrow int \rightarrow int}{\quad}} \quad \frac{\overline{\text{true}:\{\} \vdash bool}}{\quad}}{\frac{(\text{fun } x \rightarrow (\text{fun } y \rightarrow y + x)) \text{ true}:\times}{\quad}}$$

Fig. 4. Principal typing tree of $(\text{fun } x \rightarrow (\text{fun } y \rightarrow y + x)) \text{ true}$

A typing consists of a type environment Γ , a finite mapping from free variables to types, and a type τ . Instead of the more common $\Gamma \vdash M : \tau$ we write $M : \Gamma \vdash \tau$ to state that expression M has typing $\Gamma \vdash \tau$. The special typing \times indicates that no typing of the form $\Gamma \vdash \tau$ exists; so \times indicates a type error.

We obtain expected typings from principal typings. For a program a syntax tree where each node is annotated with a principal typing can be constructed¹; details are given by Chitil, Tsushima and Asai [2, 12].

Most type inference algorithms try to construct a tree based on the Hindley-Milner rules. To see how the PTT differs, consider the following ill-typed OCaml program: $(\text{fun } x \rightarrow (\text{fun } y \rightarrow y + x)) \text{ true}$

The function $+$ adds two integers; its type is $int \rightarrow int \rightarrow int$. The constant true has type $bool$. Hence the program is ill-typed.

Figure 3 shows a Hindley-Milner type inference tree and Fig. 4 shows the PTT for this program. In the PTT each subexpression appears with its principal typing. For example, in the left upper part of Fig. 4, y has type β not int because there is no constraint to have type int in y itself; also the variable x does not appear in this type environment at all. In contrast in Fig. 3, the subexpression y has the type environment $\{x: int, y: int\}$, because of constraints in other parts of the tree.

3 Expected Typings and Type Debugging Information Tree

We obtain the type debugging information tree from the PTT by adding an expected typing to every node of the tree. So in every tree node an expression is annotated with both its principal and its expected typing.

¹ It is actually a tree of principal monomorphic typings; let-bound potentially polymorphic variables do not appear in the type environment [2].

At first we focus on programs that have only a single type error. Afterwards we show why we have to obtain expected typings from principal typings, not the typings of a standard Hindley-Milner type inference tree. Finally we show how we can obtain expected typings from programs that have multiple type errors, the normal case in practice.

3.1 How to Obtain Expected Types

For simplicity we assume that we have an ill-typed program with a PTT that has only one type error node. Let us reconsider the example from the preceding section: `(fun x -> (fun y -> y + x)) true`. The PTT of this program is shown in Fig. 4.

Inference of expected typings starts from the root of the tree. We assume that the expected type of the whole program is some type variable α , which means that there is no type constraint. Because the whole program has no free variables, its type environment is empty. Our next goal is to infer the expected typings shown as black boxes below. The first box is the expected typing of the function `(fun x -> (fun y -> y + x))` and the second box is the expected typing of `true`.

$$\frac{\begin{array}{c} \text{(fun x -> (fun y -> y + x)) : } \left\{ \begin{array}{l} \{\} \vdash \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \\ \blacksquare \end{array} \right. \quad \text{true : } \left\{ \begin{array}{l} \{\} \vdash \text{bool} \\ \blacksquare \end{array} \right. \end{array}}{\text{(fun x -> (fun y -> y + x)) true : } \left\{ \begin{array}{l} \times \\ \{\} \vdash \alpha \end{array} \right.}$$

The idea for obtaining the expected typing of an expression M is that we do not use the typing of M itself but use the typing of its sibling nodes and parent node.

Here the type environments are empty and we are solely concerned with types. We obtain the expected type of `(fun x -> (fun y -> y + x))` from the principal type of the argument `true` and the expected type of the whole program. We can visualize the constraint of the function application as follows:

$$\text{(fun x -> (fun y -> y + x)) : } \blacksquare \text{ (true : bool) : } \alpha$$

Thus we see that the black box must be $\text{bool} \rightarrow \alpha$. Similarly we can obtain the expected type of `true`, namely int .

3.2 Inferring Expected Typings

Figure 5 shows our inference rules for expected typings. We assume that all principal typings and the expected typing of an expression at the bottom of a rule are known. The rules define the expected typings for the subexpressions on top of the rules. In the figure these inferred expected typings are boxed.

$$\begin{array}{c}
\overline{\mathbf{n}^l : \left\{ \begin{array}{l} (F_i \vdash \tau_i, \{l\}) \\ (F_e \vdash \tau_e, L_e) \end{array} \right\}} \qquad \overline{\mathbf{v}^l : \left\{ \begin{array}{l} (F_i \vdash \tau_i, \{l\}) \\ (F_e \vdash \tau_e, L_e) \end{array} \right\}} \\
\\
M : \left\{ \begin{array}{l} (F_{i'} \vdash \tau_{i'}, L_0) \\ \boxed{\text{mgu}(\{F_e\}, \{\tau_e = \alpha \rightarrow \beta\}, \beta), L_e \cup \{l\}} \end{array} \right\} \\
\text{fun}^l x \rightarrow M : \left\{ \begin{array}{l} (F_i \vdash \tau_i, L_0 \cup \{l\}) \\ (F_e \vdash \tau_e, L_e) \end{array} \right\} \\
\\
M_0 : \left\{ \begin{array}{l} (F_0 \vdash \tau_0, L_0) \\ \boxed{\text{mgu}(\{F_e, F_1\}, \{\tau_1 \rightarrow \tau_e\}, \tau_e), L_e \cup L_1 \cup \{l\}} \end{array} \right\} \\
M_1 : \left\{ \begin{array}{l} (F_1 \vdash \tau_1, L_1) \\ \boxed{\text{mgu}(\{F_e, F_0\}, \{\tau_0 = \alpha \rightarrow \tau_e\}, \alpha), L_e \cup L_0 \cup \{l\}} \end{array} \right\} \\
\hline
(M_0 \ M_1)^l : \left\{ \begin{array}{l} (F_i \vdash \tau_i, L_0 \cup L_1 \cup \{l\}) \\ (F_e \vdash \tau_e, L_e) \end{array} \right\} \\
\\
M_0 : \left\{ \begin{array}{l} (F_0 \vdash \tau_0, L_0) \\ \boxed{\text{mgu}(\{F_e, F_1\}, \{\tau_e = \alpha * \beta\}, \alpha), L_e \cup L_1 \cup \{l\}} \end{array} \right\} \\
M_1 : \left\{ \begin{array}{l} (F_1 \vdash \tau_1, L_1) \\ \boxed{\text{mgu}(\{F_e, F_0\}, \{\tau_e = \alpha * \beta\}, \beta), L_e \cup L_0 \cup \{l\}} \end{array} \right\} \\
\hline
(M_0, \ M_1)^l : \left\{ \begin{array}{l} (F_i \vdash \tau_i, L_0 \cup L_1 \cup \{l\}) \\ (F_e \vdash \tau_e, L_e) \end{array} \right\} \\
\\
M_0 : \left\{ \begin{array}{l} (F_0 \vdash \tau_0, L_0) \\ \boxed{\text{mgu}(\{F_1, F_2, F_e\}, \{\tau_1 = \tau_e, \tau_2 = \tau_e\}, \text{bool}), L_e \cup L_1 \cup L_2 \cup \{l\}} \end{array} \right\} \\
M_1 : \left\{ \begin{array}{l} (F_1 \vdash \tau_1, L_1) \\ \boxed{\text{mgu}(\{F_0, F_2, F_e\}, \{\tau_0 = \text{bool}, \tau_2 = \tau_e\}, \tau_e), L_e \cup L_0 \cup L_2 \cup \{l\}} \end{array} \right\} \\
M_2 : \left\{ \begin{array}{l} (F_2 \vdash \tau_2, L_2) \\ \boxed{\text{mgu}(\{F_0, F_2, F_e\}, \{\tau_0 = \text{bool}, \tau_1 = \tau_e\}, \tau_e), L_e \cup L_0 \cup L_1 \cup \{l\}} \end{array} \right\} \\
\hline
\text{if}^l M_0 \text{ then } M_1 \text{ else } M_2 : \left\{ \begin{array}{l} (F_i \vdash \tau_i, L_0 \cup L_1 \cup L_2 \cup \{l\}) \\ (F_e \vdash \tau_e, L_e) \end{array} \right\} \\
\\
M_0 : \left\{ \begin{array}{l} ((F_0, \tau_0), L_0) \\ \boxed{\text{mgu}(\{F_e, F_1, F_p\}, \{\tau_e = \tau_1\}, \tau_p) \setminus (\text{fv}(p)), L_e \cup L_1 \cup \{l\}} \end{array} \right\} \\
M_1 : \left\{ \begin{array}{l} ((F_1, \tau_1), L_1) \\ \boxed{\text{mgu}(\{F_e, F_0, F_p\}, \{\tau_0 = \tau_p\}, \tau_e), L_e \cup L_0 \cup \{l\}} \end{array} \right\} \\
\text{where } F_p \vdash \tau_p \text{ is the principal typing of } p \\
\hline
\text{match}^l M_0 \text{ with } | \ p \rightarrow M_1 : \left\{ \begin{array}{l} (F_i \vdash \tau_i, L_0 \cup L_1 \cup \{l\}) \\ (F_e \vdash \tau_e, L_e) \end{array} \right\}
\end{array}$$

Fig. 5. Inference rules for expected typings

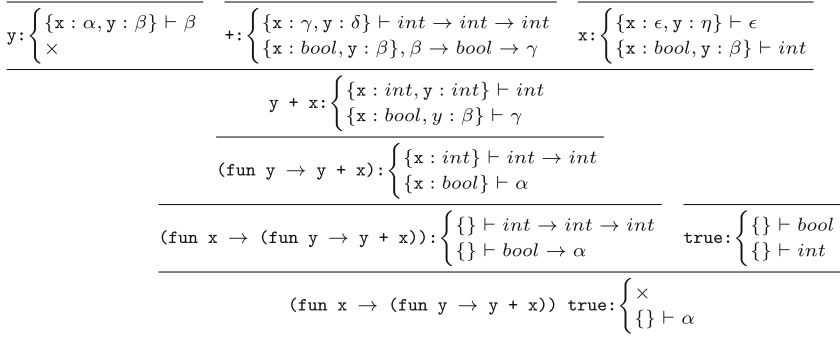


Fig. 6. Type debugging information tree of (fun x -> (fun y -> y + x)) true

For many language constructs we need to compose several type environments and solve a set of equational type constraints. Hence we define and use a most general unifier function called mgu. The function call

$$\text{mgu}(\{\Gamma_1, \dots, \Gamma_n\}, \{\tau_1 = \tau'_1, \dots, \tau_n = \tau'_n\}, \tau)$$

computes a most general type substitution σ with

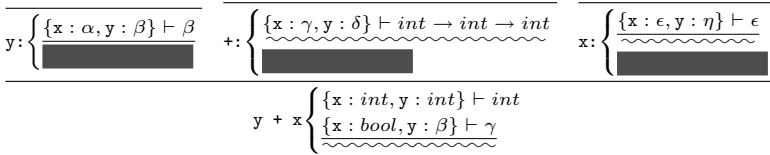
$$\begin{aligned}
 \Gamma_1 \sigma &= \dots = \Gamma_n \sigma \\
 \tau_1 \sigma &= \tau'_1 \sigma, \dots, \tau_n \sigma = \tau'_n \sigma
 \end{aligned}$$

and returns the typing

$$\Gamma_1 \sigma \vdash \tau \sigma$$

If no such substitution exists, it returns \times .

To understand the function mgu and the inference rule for application, let us consider the top of Fig. 6:



First, let us determine the expected typing of $+$. We use the underlined typings: the expected typing of $+$'s parent node $y + x$ and the principal typings of y and x . Because these type constraints must be satisfied simultaneously, their environments must be composed. Because $+$ is a function applied to the two arguments, we have the additional constraint that its expected type is $\beta \rightarrow \epsilon \rightarrow \gamma$. In summary, $+$ has the expected typing

$$\begin{aligned}
 &\text{mgu}(\{\{x : \alpha, y : \beta\}, \{x : \epsilon, y : \eta\}, \{x : bool, y : \beta\}\}, \{\}, \beta \rightarrow \epsilon \rightarrow \gamma) \\
 &= \{x : bool, y : \beta\} \vdash \beta \rightarrow bool \rightarrow \gamma
 \end{aligned}$$

Second, let us determine the expected typing of y in the top of Fig. 6. We use the curve-underlined typings: the expected typing of $y + x$ and the principal typings of $+$ and x . We need to combine the type environments of the three typings and express the constraint that $+$ is a function with y and x as arguments. In summary, y has the expected typing

$$\begin{aligned} & \text{mgu}(\{\{x : \gamma, y : \delta\}, \{x : \epsilon, y : \eta\}, \\ & \quad \{x : \text{bool}, y : \beta\}\}, \{int \rightarrow int \rightarrow int = \kappa \rightarrow \epsilon \rightarrow \gamma\}, \kappa) \\ & = \times \end{aligned}$$

We obtain \times , because there does not exist any unifying substitution.

Nearly all our inference rules in Fig. 5 use the mgu function. The only exception are the rules for constants and variables, because they have no smaller components; their own expected typings are determined by their contexts. The pattern match construct binds new variables in the pattern p . Hence these variables have to be removed from the typing for the term M_0 .

3.3 Type Annotations/Signatures

Unlike all real functional programming languages, our core language does not have type annotations, which allows the user to specify that a function should have a certain type. However, if we do consider type annotations, then these naturally contribute to our expected typings. Expected typings propagate the user’s annotations towards the leaves of the type debugging information tree. Thus expected typings become more informative and we also have more expected typings \times .

3.4 Why Obtain Expected Typings from Principal Typings?

We cannot obtain expected typings by means of the standard type inference tree. Consider again the expression $(\text{fun } x \rightarrow (\text{fun } y \rightarrow y + x)) \text{ true}$. We obtain the following tree from the Hindley-Milner type rules:

$$\frac{\frac{y: \left\{ \begin{array}{l} \{x : int, y : int\} \vdash int \\ \blacksquare \end{array} \right.}{\quad} \quad \frac{+ : \left\{ \begin{array}{l} \{x : int, y : int\} \vdash int \rightarrow int \rightarrow int \\ \blacksquare \end{array} \right.}{\quad} \quad \frac{x : \left\{ \begin{array}{l} \{x : int, y : int\} \vdash int \\ \blacksquare \end{array} \right.}{\quad}}{y + x : \left\{ \begin{array}{l} \{x : int, y : int\} \vdash int \\ \{x : bool, y : \beta\} \vdash \gamma \end{array} \right.}}$$

We want to determine the expected typing of $+$. However, if we now compose the type environments for y , x and $y + x$, we get

$$\begin{aligned} & \text{mgu}(\{\{x : int, y : int\}, \{x : int, y : int\}, \{x : bool, y : \beta\}\}, \{\}, int \rightarrow int \rightarrow \gamma) \\ & = \times \end{aligned}$$

because the types of x in the type environments are already in conflict. We cannot use the standard type inference tree, because type information in type environments includes type constraints of many parts of the program.

3.5 Expected Typings in the Presence of Multiple Type Errors

If a program has only a single type error, then the expected typing of an expression is derived from all other parts of the program except the expression. In the presence of multiple type errors, we should not derive the expected typing of an expression from all other parts, because these other parts contain type errors and hence many typings are \times . Note that even in the presence of a single type error several principal and expected typings are already \times , but we simply exclude these when inferring expected typings. However, if we applied the same method in the presence of multiple type errors, then we would lose too much type information and our expected typings would become useless.

To control which type constraints to include and which not, we need to record the program parts that contribute to the expected typings, which we represent with the notation $(\Gamma \vdash \tau, \{l_1, \dots, l_n\})$. This means that typing $\Gamma \vdash \tau$ is derived from type information of the subexpressions with the locations l_1, \dots, l_n .

Let us consider the following multiple type error example: $((x\ 1, y\ 2), (x\ \text{false}, y\ \text{true}))$. This program includes multiple type errors, because there are two type conflicts about x and y . Let us focus on $(x\ 1, y\ 2)$ in this program. In the first step we obtain the following PTT.

$$\frac{x\ 1 : (\{x : \text{int} \rightarrow \alpha\} \vdash \alpha, L_0) \quad y\ 2 : (\{y : \text{int} \rightarrow \beta\} \vdash \beta, L_1)}{(x\ 1, y\ 2)^t : (\{x : \text{int} \rightarrow \alpha; y : \text{int} \rightarrow \beta\} \vdash \alpha * \beta, L_0 \cup L_1 \cup \{l\})}$$

In the second step we obtain the following tree by the rules in Fig. 5.

$$\frac{x\ 1 : \begin{cases} (\{x : \text{int} \rightarrow \alpha\} \vdash \alpha, L_0) \\ \times \end{cases} \quad y\ 2 : \begin{cases} (\{y : \text{int} \rightarrow \beta\} \vdash \beta, L_1) \\ \times \end{cases}}{(x\ 1, y\ 2)^t : \begin{cases} (\{x : \text{int} \rightarrow \alpha; y : \text{int} \rightarrow \beta\} \vdash \alpha * \beta, L_0 \cup L_1 \cup \{l\}) \\ \boxed{(\{x : \text{bool} \rightarrow \gamma; y : \text{bool} \rightarrow \delta\} \vdash \epsilon, L_e)} \end{cases}}$$

The expected typing of $(x\ 1, y\ 2)$ is obtained from its sibling $(x\ \text{false}, y\ \text{true})$'s principal typing. We successfully obtain the expected typing for $(x\ 1, y\ 2)$ (the boxed part). Because the principal typings of $x\ 1$ and $y\ 2$ have conflicts with the expected typing of their parent node, each child has expected typing \times . We need more expected typings that we could not obtain in this step. In the third step, we reconstruct the following abstracted PTT using the upper tree information.

$$\frac{x\ 1 : (\{\} \vdash \alpha', \{\}) \quad y\ 2 : (\{\} \vdash \beta', \{\})}{(x\ 1, y\ 2)^t : (\{\} \vdash \alpha' * \beta', \{\})}$$

The reconstruction rules of PTT is the following. If an expression has an expected typing \times in the past steps, we put the abstracted typing $(\{\} \vdash \gamma, \{\})$ (γ is a new type variable that does not appear anywhere. The second $\{\}$ means that this typing does not include any constraints) for it. Otherwise we use standard method in [2, 12] for obtaining their principal typings. So in this case, the principal typings of $x\ 1$ and $y\ 2$ are abstracted typings. On the other hand, the principal typing of $(x\ 1, y\ 2)$ is not abstracted and inferred by its children. This is because $(x\ 1, y\ 2)$ has expected typing in the second step. In the fourth step we apply our rules in Fig. 5 to this tree and obtain the following tree.

$$\begin{array}{c}
\mathbf{x} \ 1 : \left\{ \begin{array}{l} (\{\} \vdash \alpha', \{\}) \\ \boxed{(\{\mathbf{x} : \mathit{bool} \rightarrow \gamma; \mathbf{y} : \mathit{bool} \rightarrow \delta\}, \zeta, L_e \cup \{l\})} \end{array} \right. \\
\mathbf{y} \ 2 : \left\{ \begin{array}{l} (\{\} \vdash \beta', \{\}) \\ \boxed{(\{\mathbf{x} : \mathit{bool} \rightarrow \gamma; \mathbf{y} : \mathit{bool} \rightarrow \delta\}, \eta, L_e \cup \{l\})} \end{array} \right. \\
\hline
(\mathbf{x} \ 1, \mathbf{y} \ 2)^t : \left\{ \begin{array}{l} (\{\} \vdash \alpha' * \beta', \{l\}) \\ (\{\mathbf{x} : \mathit{bool} \rightarrow \gamma; \mathbf{y} : \mathit{bool} \rightarrow \delta\} \vdash \epsilon, L_e) \end{array} \right.
\end{array}$$

In the fourth step we could obtain the expected typings of $\mathbf{x} \ 1$ and $\mathbf{y} \ 2$ (the boxed parts). Thanks to these expected typings, we can obtain their children's expected typings subsequently. Finally we can obtain the following type debugging information tree.

$$\begin{array}{c}
\mathbf{x} \ 1 : \left\{ \begin{array}{l} (\{\mathbf{x} : \mathit{int} \rightarrow \alpha\} \vdash \alpha, L_0) \\ (\{\mathbf{x} : \mathit{bool} \rightarrow \gamma; \mathbf{y} : \mathit{bool} \rightarrow \delta\}, \zeta, L_e \cup \{l\}) \end{array} \right. \\
\mathbf{y} \ 2 : \left\{ \begin{array}{l} (\{\mathbf{y} : \mathit{int} \rightarrow \beta\} \vdash \beta, L_1) \\ (\{\mathbf{x} : \mathit{bool} \rightarrow \gamma; \mathbf{y} : \mathit{bool} \rightarrow \delta\}, \eta, L_e \cup \{l\}) \end{array} \right. \\
\hline
(\mathbf{x} \ 1, \mathbf{y} \ 2)^t : \left\{ \begin{array}{l} (\{\mathbf{x} : \mathit{int} \rightarrow \alpha; \mathbf{y} : \mathit{int} \rightarrow \beta\} \vdash \alpha * \beta, L_0 \cup L_1 \cup \{l\}) \\ (\{\mathbf{x} : \mathit{bool} \rightarrow \gamma; \mathbf{y} : \mathit{bool} \rightarrow \delta\} \vdash \epsilon, L_e) \end{array} \right.
\end{array}$$

The algorithm in presence of multiple type errors is the following.

1. (Re)construct PTT. If the expected typing of an expression in the past steps is \times , we use abstracted typing $(\{\} \vdash \gamma, \{\})$ (γ is a new type variable that does not appear anywhere). Otherwise we use standard method to infer its principal typing in [2,12].
2. By the rules in Fig. 5. Infer expected typings that we could not obtain yet.
3. If we obtain new expected typings in step 2, we repeat step 1. Otherwise construction of type debugging information tree is finished.

For doing this, we need a restriction: each tree node has at most two children. Thanks to this restriction we can determine if the sibling node information is needed or not. So before inferring principal and expected typings we transform the tree such that every node has at most two children.

4 Using the Type Debugging Information Tree

In this section we show how the type debugging information tree and its typings can be used to develop different type debugging tools.

4.1 Enumeration of Type Error Messages

The type error messages of a compiler are most familiar to programmers. For an ill-typed program the OCaml compiler stops after producing one type error message. From our type error debugging tree we can easily produce many type error messages: Every tree node where the principal typing and expected typing are in conflict, that is, cannot be unified, is a type error node and thus yields a type error message.

For example, the tree in Fig. 1 yields four error messages about leaves ($\hat{_}$, `fst`, `2`, `:: (in [2])`) and six error messages about inner nodes.

For each type error node we can produce a type error message as follows:

```
(fst  $\hat{\_}$  n)
This expression has type string -> string -> string
but an expression was expected of type int -> 'b -> 'c
```

In practice the order in which the type error messages are shown is important. The most likely causes, based on some heuristics, should be shown first [1].

4.2 Type Error Slicing

A type error slice is a slice of a program that on its own is ill-typed. A type error slicer receives an ill-typed program and returns one or many type error slices. For our example

```
(fun x -> (fun y -> y + x)) true
```

the type debugging information tree enables us to produce the type error slice

```
(fun x -> (fun y -> .. + x)) true
```

In a slice `..` denotes any subexpression that does not belong to the slice.

We obtained the type error slice by simply removing any subexpression that has an expected typing \times , here just the subexpression `y`. An expected typing \times indicates that the types of the surrounding program, from which the expected typing is determined, are already in conflict. This simple method works well when there is just a single type error slice. When there are several type errors, we can potentially obtain many slices using the following algorithm:

1. Each node of the type debugging information tree has
 - a location l of the programming construct of the node,
 - a set of locations L_i from which the principal typing was determined
 - a set of locations L_e from which the expected typing was determined.
2. We remove elements in L_i and L_e that are not needed in opposite directions, obtaining smaller sets $L'_i \subseteq L_i$ and $L'_e \subseteq L_e$.
3. If $\{l\} \cup L'_i \cup L'_e$ is not subset of a type error slice that we already have, it is a new type error slice.

4.3 Improved Interactive Type Error Debugging

An interactive type debugger asks the user for information to determine the source of a type error. The PTT was originally defined to support algorithmic debugging of type errors. Here is a short example session, with the user's input underlined, demonstrating algorithmic debugging using the PTT of Fig. 1:


```

1. Is your intended type of f [string] -> string -> [string]?
> No
...
6. Is your intended type of ^ string -> string -> string?
> No
Type error debugger locates the source of a type error: ^
Against intentions, its type is string -> string -> string

```

The source of the type error is located after six questions.

The questions of algorithmic debugging are based on a walk through the PTT. The session starts at a node that has a principal typing \times , but all its child nodes have a principal typing unequal \times . Here the session starts at the root of the PTT.

Using the type error information tree, we start at a type error node instead. Because leaves of the tree are often sources of type errors and their typings are easier to understand, we start at a leaf type error node. So in our example we might start with the node for \wedge and thus successfully finish algorithmic debugging after one question.

Adding type annotations (signatures) to a program adds information to the expected typings. More of them become \times , which excludes the nodes from being asked about in an algorithmic debugging session. If in our example we add that f should have type $[int] \rightarrow int \rightarrow [int]$, then \wedge is the only leaf type error node of the type debugging information tree and hence the algorithmic debugging session has to start with it.

5 Evaluation

We evaluate our prototype using fifteen programs, each of which has one type error. Most of them are from the functional programming courses in Ochanomizu University. Some examples are from the online demonstration of Skalpel [13]. Three of them include multiple type conflicts. The OCaml compiler's error message locates the cause of a type error correctly for Test 5, 7, 9 and 11.

We rank error messages by giving higher priority to leaves than inner nodes, and higher priority to large location sets.

How Do Expected Typings Reduce the Search Space of Type Debugging? Figure 7 shows the number of lines of each program and its number of expected typings. The sum of the expected typings are one third of the sum of the program sizes. Basically large programs have more locations that do not contribute to type errors; therefore expected typing will be more effective in large programs.

Are User's Type Annotation Useful? The Fig. 8 shows the number of expected typings of leaves and inner nodes, respectively. The blue line and green line show the expected typings without user's annotations. The other two lines are with user's annotations. From this figure we see that user's annotations

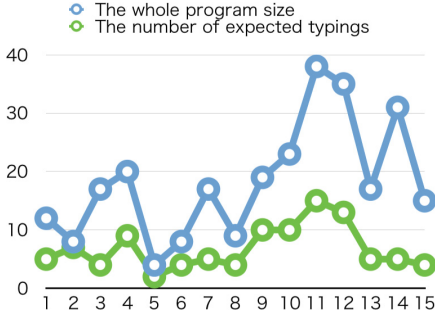


Fig. 7. The program sizes and the number of expected typings

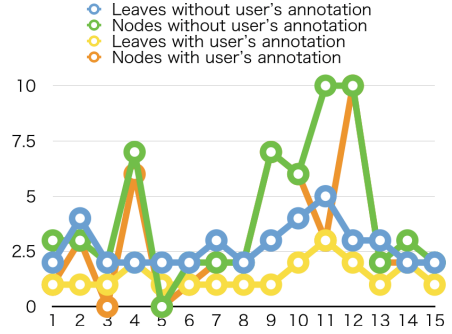


Fig. 8. The number of expected typings (leaves and inner nodes) (Color figure online)

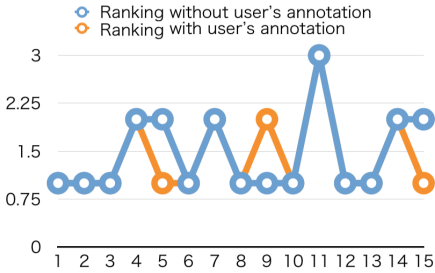


Fig. 9. The numbers of question to answer (without and with user's annotation)

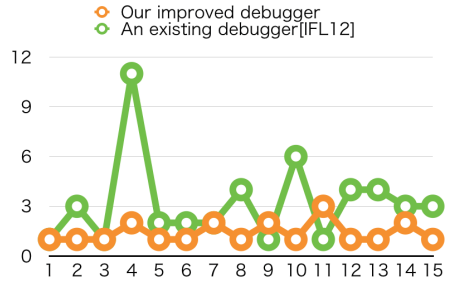


Fig. 10. Comparison with the existing debugger

reduce the search space of type debugging. This reduction could not be achieved without expected type inference.

How Many Questions Does Algorithmic Debugging Ask? Figure 9 shows the numbers of questions until we locate the type error source. Because the numbers are small, our strategy looks effective for type debugging. There are few expected typings; therefore there is no significant difference.

How Much Does Our System Improve Interactive Debugging? In Fig. 10 we compare our debugger (adding a type annotation for the whole program) with an existing implementation [12] (for convenience we call this the IFLdebugger). In most cases our debugger can locate the source of a type error faster than the IFLdebugger. In some cases the IFLdebugger locates faster; however, this depends on the bias, from the bottom of the tree. Especially in the cases that the source of a type error is far from the type conflicted part (starting point of debugging) our method is effective (Test 4 and Test 10).

6 Related Work

New Type Inference Algorithms. Many researchers designed new type inference algorithms which are better than Milner’s algorithm \mathcal{W} for type error debugging. For example, Lee and Yi [6] presented algorithm \mathcal{M} which finds type conflicts earlier than algorithms \mathcal{W} . Like \mathcal{W} , algorithm \mathcal{M} is biased: when there is a type conflict between two subexpressions in a program, it always blames the subexpression on the right. Both algorithms stop at the first type conflict that they find. In contrast, Neubauer and Thiemann [7], and Chen and Erwig [1] define type inference algorithms that succeed for any program. They extend the type system in different ways to allow an expression to have multiple types. Otherwise ill-typed expressions are typed with the new ‘multi-types’. As part of presenting type debugging information to the user, Chen and Erwig formalise the notion of expected type.

Interactive Type Error Debugging. Chitil [2] emphasises that the cause of a type error depends on the intentions of the programmer. Hence type error debugging should be an interactive process involving the programmer. He shows how to apply algorithmic debugging [9] to type error debugging: the programmer has to answer a series of questions by the debugger. Chitil defines the PTT as foundation for algorithmic debugging. Tsushima and Asai [11] implement a type debugger for OCaml based on Chitil’s idea.

Type Inference as Constraint Solving Problem. Every programming language construct puts a constraint on the type of itself and its direct subexpressions. So a program can first be translated into a set of constraints which are solved in a separate phase. A minimal unsatisfiable subset of constraints explains a type error. If each constraint is associated with the program locations that gave rise to it, then a minimal unsatisfiable subset of constraints defines a slice of the program that explains the type error to the user. Heeren et al. [4] annotate a syntax tree of the program with type constraints such that different algorithms can solve these constraints in different orders. The Chameleon system [10] fully implemented the type-inference-as-constraint-solving approach for an expressive constraint language that supports many extensions of the Hindley-Milner type system, especially Haskell’s classes. Besides presenting slices, Chameleon also provides interactive type error debugging. Concurrently Haack and Wells [3] applied the same idea to the Hindley-Milner type system. They later developed their method into the tool Skalpel for type error slicing of ML programs [13].

Reusing the Compiler’s Type Inference as Black Box. Instead of inventing a new type inference algorithm, several researchers developed methods for reusing the existing type inference implementation of a compiler for type error debugging. The central motivation is that implementing type inference for a real programming language is a major investment already made; besides, the type systems of some languages such as Haskell continuously evolve. Schilling [8] developed a type error slicer for a subset of Haskell that produces program slices similar to Skalpel. Tsushima and Asai [12] built an interactive debugger for OCaml that provides the user experience of Chitil’s approach (cf. Sect. 4.3).

Comparison. To produce the PTT of a program, our implementation uses the method of Tsushima and Asai, reusing the existing compiler’s type inference. Hence, even though we have to add expected typings, unlike many other methods we do not require a reimplementa-tion of type inference. Our framework supports several type debugging approaches with different user experiences, including the program slices of constraint solving methods and interactive type error debugging. Although constraints support formalising type systems and type debugging, they are not directly suitable for being shown to the user.

7 Conclusion

In this paper we argue that expected typings, which provide type information from the context of an expression, are useful for type error debugging. We describe a common framework for several type error debugging approaches. Our prototype supports the following claims

- Expected typings reduce the search space of type debugging.
- Propagating user’s type annotations additionally reduces the search space of type debugging.

Our framework allows for a synergy of previously independent methods than with existing type debuggers.

When a program contains many type errors, our expected typings are often not informative enough. We believe that this is because we use a coarse typing \times to express any kind of type conflict. In the future we want to explore how we can extend our framework with a more fine-grained language of typings, where the language of types is extended by a construct \times and instead of a typing \times we have a typing like $\{x : \times, y : \beta\} \vdash \kappa$.


References

1. Chen, S., Erwig, M.: Counter-factual typing for debugging type errors. In: POPL 2014, pp. 583–594 (2014)
2. Chitil, O.: Compositional explanation of types and algorithmic debugging of type errors. In: ICFP 2001, pp. 193–204 (2001)
3. Haack, C., Wells, J.B.: Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.* **50**(1–3), 189–224 (2004). Special issue ESOP 2003
4. Heeren, B., Hage, J., Swierstra, S.: Constraint based type inferencing in Helium. In: Immediate Applications of Constraint Programming (ACP), pp. 57–78 (2003)
5. Ishii, Y., Asai, K.: Report on a user test and extension of a type debugger for novice programmers. In: Post Conference Proceedings of International Workshop on Trends in Functional Programming in Education, pp. 1–18 (2014)
6. Lee, O., Yi, K.: Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.* **20**, 707–723 (1998)
7. Neubauer, M., Thiemann, P.: Discriminative sum types locate the source of type errors. In: ICFP 2003, pp. 15–26 (2003)

8. Schilling, T.: Constraint-free type error slicing. In: Peña, R., Page, R. (eds.) TFP 2011. LNCS, vol. 7193, pp. 1–16. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32037-8_1
9. Shapiro, E.Y.: Algorithmic Program Debugging. MIT Press, Cambridge (1983)
10. Stuckey, P.J., Sulzmann, M., Wazny, J.: Interactive type debugging in Haskell. In: Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell 2003), pp. 72–83 (2003)
11. Tsushima, K., Asai, K.: Report on an OCaml type debugger. In: ML Workshop, 3 pages (2011)
12. Tsushima, K., Asai, K.: An embedded type debugger. In: Hinze, R. (ed.) IFL 2012. LNCS, vol. 8241, pp. 190–206. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41582-1_12
13. Rahli, V., Wells, J., Pirie, J., Kamareddine, F.: Skalpel. *J. Symb. Comput.* **80**(P1), 164–208 (2017)



CauDER: A Causal-Consistent Reversible Debugger for Erlang

Ivan Lanese¹ , Naoki Nishida² , Adrián Palacios³ ,
and Germán Vidal³  

¹ Focus Team, University of Bologna/Inria, Bologna, Italy
ivan.lanese@gmail.com

² Graduate School of Informatics, Nagoya University, Nagoya, Japan
nishida@i.nagoya-u.ac.jp

³ MiST, DSIC, Universitat Politècnica de València, Valencia, Spain
{[apalacios](mailto:apalacios@dsic.upv.es),[gvidal](mailto:gvidal@dsic.upv.es)}@dsic.upv.es

Abstract. Programming languages based on the actor model, such as Erlang, avoid some concurrency bugs by design. However, other concurrency bugs, such as message order violations and livelocks, can still show up in programs. These hard-to-find bugs can be more easily detected by using causal-consistent reversible debugging, a debugging technique that allows one to traverse a computation both forward and backward. Most notably, causal consistency implies that, when going backward, an action can only be undone provided that its consequences, if any, have been undone beforehand. To the best of our knowledge, we present the first causal-consistent reversible debugger for Erlang, which may help programmers to detect and fix various kinds of bugs, including message order violations and livelocks.

1 Introduction

Over the last years, concurrent programming has become a common practice. However, it is also a difficult and error-prone activity, since concurrency enables faulty behaviours, such as *deadlocks* and *livelocks*, which are hard to avoid, detect and fix. One of the reasons for these difficulties is that these behaviours may show up only in some extremely rare circumstances (e.g., for some unusual scheduling).

A recent analysis [16] reveals that most of the approaches to software validation and debugging in message-passing concurrent languages like Erlang are

This work has been partially supported by MINECO/AEI/FEDER (EU) under grant TIN2016-76843-C4-1-R, by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic), by COST Action IC1405 on Reversible Computation - extending horizons of computing, and by JSPS KAKENHI Grant Number JP17H01722.

A. Palacios—Partially supported by the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores* and *Ayudas a la movilidad predoctoral para la realización de estancias breves en centros de I+D*, MINECO (SEIDI), under FPI grants BES-2014-069749 and EEBB-I-16-11469.

based on some form of static analysis (e.g., Dialyzer [15], McErlang [6], Soter [5]) or testing (e.g., QuickCheck [3], PropEr [18], Concuerror [10], CutEr [9]). However, these techniques are helpful only to find some specific categories of problems. On the other hand, traditional debuggers (like the one included in the OTP Erlang distribution) are sometimes not particularly useful when an unusual interleaving brings up an error, since recompiling the program for debugging may give rise to a completely different execution behaviour. In this setting, *causal-consistent reversible debugging* [7] may be useful to complement the previous approaches. Here, one can run a program in the debugger in a controlled manner. If something (potentially) incorrect shows up, the user can stop the forward computation and go backwards—in a causal-consistent way—to look for the origin of the problem. In this context, we say that a backward step is *causal consistent* [4, 12] if an action cannot be undone until all the actions that depend on it have already been undone. Causal-consistent reversibility is particularly relevant for debugging because it allows us to undo the actions of a given process in a stepwise manner while ignoring the actions of the remaining processes, unless they are causally related. In a traditional reversible debugger, one can only go backwards in exactly the reverse order of the forward execution, which makes focusing on undoing the actions of a given process much more difficult, since they can be interleaved with completely unrelated actions from other processes.

The main contributions of this paper are the following. We have designed and implemented CauDEr, a publicly available software tool for causal-consistent reversible debugging of (a subset of) Erlang programs. The tool builds upon some recent developments on the causal-consistent reversible semantics of Erlang [13, 17], though we also introduce (in Sect. 3) a new rollback semantics which is especially tailored for reversible debugging. In this semantics, one can for instance run a program backwards up to the sending of a particular message, the creation of a given process, or the introduction of a binding for some variable. We present our tool and illustrate its use for finding bugs that would be difficult to deal with using the previously available tools (Sect. 4). We use a concurrent implementation of the dining philosophers problem as a running example. CauDEr is publicly available from <https://github.com/mistupv/cauder>.

2 The Language

Erlang is a message passing concurrent and distributed functional programming language. We define our technique for (a subset of) Core Erlang [2], which is used as an intermediate representation during the compilation of Erlang programs. In this section, we describe the syntax and semantics of the subset of Core Erlang we are interested in.

The syntax of the language can be found in Fig. 1. A module is a sequence of function definitions, where each function name f/n (atom/arity) has an associated definition of the form `fun` $(X_1, \dots, X_n) \rightarrow e$. We consider that a program consists of a single module for simplicity. The body of a function is an *expression*, which can include variables, literals, function names, lists, tuples, calls to built-in

$$\begin{aligned}
 \text{module} &::= \text{module } Atom = fun_1, \dots, fun_n \\
 \text{fun} &::= \text{fname} = \text{fun } (X_1, \dots, X_n) \rightarrow \text{expr} \\
 \text{fname} &::= Atom/Integer \\
 \text{lit} &::= Atom \mid Integer \mid Float \mid [] \\
 \text{expr} &::= Var \mid lit \mid \text{fname} \mid [expr_1|expr_2] \mid \{expr_1, \dots, expr_n\} \\
 &\quad \mid \text{call } \text{expr } (expr_1, \dots, expr_n) \mid \text{apply } \text{expr } (expr_1, \dots, expr_n) \\
 &\quad \mid \text{case } \text{expr} \text{ of } clause_1; \dots; clause_m \text{ end} \\
 &\quad \mid \text{let } Var = \text{expr}_1 \text{ in } \text{expr}_2 \mid \text{receive } clause_1; \dots; clause_n \text{ end} \\
 &\quad \mid \text{spawn}(\text{expr}, [expr_1, \dots, expr_n]) \mid \text{expr}_1 ! \text{expr}_2 \mid \text{self}() \\
 \text{clause} &::= \text{pat when } \text{expr}_1 \rightarrow \text{expr}_2 \\
 \text{pat} &::= Var \mid lit \mid [pat_1|pat_2] \mid \{pat_1, \dots, pat_n\}
 \end{aligned}$$

Fig. 1. Language syntax rules

functions—mainly arithmetic and relational operators—, function applications, case expressions, let bindings, and receive expressions; furthermore, we also consider the functions `spawn`, “!” (for sending a message), and `self()` that are usually considered built-ins in the Erlang language. As is common practice, we assume that X is a fresh variable in a let binding of the form `let $X = \text{expr}_1$ in expr_2` .

In this language, we distinguish expressions, patterns, and values. In contrast to expressions, *patterns* are built from variables, literals, lists, and tuples. Finally, *values* are built from literals, lists, and tuples, i.e., they are *ground* (without variables) patterns. Expressions are denoted by e, e', e_1, e_2, \dots , patterns by $pat, pat', pat_1, pat_2, \dots$ and values by v, v', v_1, v_2, \dots . Atoms are written in roman letters, while variables start with an uppercase letter. A *substitution* θ is a mapping from variables to expressions, and $\text{Dom}(\theta) = \{X \in \text{Var} \mid X \neq \theta(X)\}$ is its domain. Substitutions are usually denoted by sets of bindings like, e.g., $\{X_1 \mapsto v_1, \dots, X_n \mapsto v_n\}$. Substitutions are extended to morphisms from expressions to expressions in the natural way. The identity substitution is denoted by *id*. Composition of substitutions is denoted by juxtaposition, i.e., $\theta\theta'$ denotes a substitution θ'' such that $\theta''(X) = \theta'(\theta(X))$ for all $X \in \text{Var}$.

In a case expression “`case e of pat_1 when $e_1 \rightarrow e'_1$; ...; pat_n when $e_n \rightarrow e'_n$ end`”; we first evaluate e to a value, say v ; then, we find (if it exists) the first clause pat_i when $e_i \rightarrow e'_i$ such that v matches pat_i (i.e., there exists a substitution σ for the variables of pat_i such that $v = pat_i\sigma$) and $e_i\sigma$ —the *guard*—reduces to *true*; then, the case expression reduces to $e'_i\sigma$. Note that guards can only contain calls to built-in functions (typically, arithmetic and relational operators).

Concurrent Features. In this work, we consider that a *system* is a pool of processes that can only interact through message sending and receiving (i.e., there is no shared memory). Each process has an associated *pid* (process identifier), which is unique in a system. Here, pids are ordinary values. Formally, a process is denoted by a tuple $\langle p, (\theta, e), q \rangle$ where p is the pid of the process, (θ, e) is the control—which consists of an environment (a substitution) and an expression to

be evaluated—and q is the process’ mailbox, a FIFO queue with the sequence of messages that have been sent to the process.

A running *system*, which we denote by $\Gamma; \Pi$, is composed by Γ , the *global mailbox*, which is a multiset of pairs of the form $(target_process_pid, message)$, and Π , which is a pool of processes. Π is denoted by an expression of the form

$$\langle p_1, (\theta_1, e_1), q_1 \rangle \mid \cdots \mid \langle p_n, (\theta_n, e_n), q_n \rangle$$

Here, “ \mid ” denotes an associative and commutative operator. We typically denote a system by an expression of the form $\Gamma; \langle p, (\theta, e), q \rangle \mid \Pi$ to point out that $\langle p, (\theta, e), q \rangle$ is an arbitrary process of the pool. Intuitively, Γ stores messages after they are sent, and before they are inserted in the target mailbox. Here, Γ (which is similar to the “ether” in [21]) is an artificial device used in our semantics to guarantee that all admissible message interleavings can be modelled.

In the following, we denote by $\overline{o_n}$ a sequence of syntactic objects o_1, \dots, o_n for some n .

The functions with side effects are `self()`, “`!`”, `spawn`, and `receive`. The expression `self()` returns the pid of a process, while `p!v` sends a message v to the process with pid p . New processes are spawned with a call of the form `spawn(a/n, [$\overline{v_n}$])`, so that the new process begins with the evaluation of `apply a/n ($\overline{v_n}$)`. Finally, an expression “`receive patn when $e_n \rightarrow e'_n$ end`” traverses the messages in the process’ queue until one of them matches a branch in the receive statement; i.e., it should find the *first* message v in the process’ queue (if any) such that `case v of pat1 when $e_1 \rightarrow e'_1; \dots; pat_n$ when $e_n \rightarrow e'_n$ end` can be reduced; then, the receive expression evaluates to the same expression to which the above case expression would be evaluated, with the additional side effect of deleting the message v from the process’ queue. If there is no matching message in the queue, the process *suspends* its execution until a matching message arrives.

Figure 2 shows an Erlang program implementing a simple client-server scheme with one server and two clients (a), as well as its translation into Core Erlang (b), where $_C$, $_X$ and $_Y$ are anonymous variables introduced during the translation process to represent sequences of actions using let expressions. The execution starts with a call to function `main/0`. It first spawns two processes that execute functions `server/0` and `client/1`, respectively, and then calls to function `client/1` too. Client requests have the form $\{P, req\}$, where P is the pid of the client. The server receives the message, returns a message `ack` to the client, and calls to function `server/0` again in an endless loop. After processing the two requests, the server will suspend waiting for another request.

Following [13], the semantics of the language is defined in a modular way, so that the labelled transition relation $\xrightarrow{\ell}$ models the evaluation of *expressions* and \hookrightarrow models the reduction of *systems*. Relation $\xrightarrow{\ell}$ follows a typical call-by-value semantics for side-effect free expressions;¹ in this case, reduction steps are labelled with τ . For the remaining functions, the expression rules cannot

¹ Because of lack of space, we are not presenting the rules of $\xrightarrow{\ell}$ here, but refer the interested reader to [13].

<pre> main() -> S = spawn(server/0, []), spawn(client/1, [S]), client(S). server() -> receive {P, req} -> P ! ack, server() end. client(S) -> S ! {self(), req}, receive ack -> ok end. </pre>	<pre> main/0 = fun () -> let S = spawn(server/0, []) in let _C = spawn(client/0, [S]) in apply client/0 (S) server/0 = fun () -> receive {P, req} -> let _X = P ! ack in apply server/0 () end client/1 = fun (S) -> let _Y = S ! {self(), req} in receive ack -> ok end </pre>
(a) Erlang	(b) Core Erlang

Fig. 2. A simple client server

complete the reduction of an expression since some information is not *locally* available. In these cases, the steps are labelled with the information needed to complete the reduction within the system rules of Fig. 3. For sending a message, an expression $p''!v$ is reduced to v with the side-effect of (eventually) storing the message v in the mailbox of process p'' . The associated label is thus $\text{send}(p'', v)$ so that rule *Send* can complete the step by adding the pair (p'', v) to the global mailbox Γ .

The remaining functions, *receive*, *spawn* and *self*, are reduced to a fresh distinguished symbol κ (a sort of *future*) in the expression rules, since the value cannot be determined locally. Therefore, in these cases, the labels also include κ . Then, the system rules of Fig. 3 will bind κ to its correct value: the selected expression in rule *Receive* and a pid in rules *Spawn* and *Self*.

To be more precise, for a receive statement, the label has the form $\text{rec}(\kappa, \overline{cl}_n)$ where \overline{cl}_n are the clauses of the receive statement. In rule *Receive*, the auxiliary function *matchrec* is used to find the first message in the queue that matches a clause, then returning a triple with the matching substitution θ_i , the selected branch e_i and the selected message v . Here, $q \setminus v$ denotes a new queue that results from q by removing the oldest occurrence of message v .

For a spawn, the label has the form $\text{spawn}(\kappa, a/n, [\overline{v}_n])$, where a/n and $[\overline{v}_n]$ are the arguments of spawn. Rule *Spawn* then adds a new process with a fresh pid p' initialised with the application $\text{apply } a/n (v_1, \dots, v_n)$ and an empty queue.

For a self, only κ is needed in the label. Rule *Self* then proceeds in the obvious way by binding κ to the pid of the process.

The rules presented so far allow one to store messages in the global mailbox, but not to deliver them. This is the task of the scheduler, which is modelled by rule *Sched*. This rule nondeterministically chooses a pair (p, v) in the global mailbox Γ and delivers the message v to the target process p . Note also that Γ is a multiset, so we use “ \cup ” as multiset union.

$$\begin{array}{l}
(\text{Seq}) \quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, (\theta, e), q \rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e'), q \rangle \mid \Pi} \\
(\text{Send}) \quad \frac{\theta, e \xrightarrow{\text{send}(p'', v)} \theta', e'}{\Gamma; \langle p, (\theta, e), q \rangle \mid \Pi \hookrightarrow \Gamma \cup (p'', v); \langle p, (\theta', e'), q \rangle \mid \Pi} \\
(\text{Receive}) \quad \frac{\theta, e \xrightarrow{\text{rec}(\kappa, \overline{cl}_n)} \theta', e' \quad \text{matchrec}(\overline{cl}_n, q) = (\theta_i, e_i, v)}{\Gamma; \langle p, (\theta, e), q \rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta' \theta_i, e' \{ \kappa \mapsto e_i \}), q \setminus v \rangle \mid \Pi} \\
(\text{Spawn}) \quad \frac{\theta, e \xrightarrow{\text{spawn}(\kappa, a/n, [\overline{v}_n])} \theta', e' \quad p' \text{ is a fresh pid}}{\Gamma; \langle p, (\theta, e), q \rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e' \{ \kappa \mapsto p' \}), q \rangle \mid \langle p', (id, \text{apply } a/n (\overline{v}_n)), [] \rangle \mid \Pi} \\
(\text{Self}) \quad \frac{\theta, e \xrightarrow{\text{self}(\kappa)} \theta', e'}{\Gamma; \langle p, (\theta, e), q \rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e' \{ \kappa \mapsto p \}), q \rangle \mid \Pi} \\
(\text{Sched}) \quad \frac{}{\Gamma \cup \{ (p, v) \}; \langle p, (\theta, e), q \rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta, e), v : q \rangle \mid \Pi}
\end{array}$$

Fig. 3. Standard semantics: system rules

3 Causal-Consistent Reversible Debugging

In this section, we present a causal-consistent reversible semantics for the considered language. The semantics is based on the reversible semantics for Erlang introduced in [13, 17]. In particular, [13] presents an *uncontrolled* reversible semantics, which is highly non-deterministic, and a *controlled* semantics that performs a backward computation up to a given *checkpoint* in a mostly deterministic way. Here, we build on the uncontrolled semantics, and define a new controlled semantics which is more appropriate as a basis for a causal-consistent reversible debugger than the one in [13].

First, following [13], we introduce an instrumented version of the standard semantics. For this purpose, we exploit a typical Landauer’s embedding [11] and include a “history” h in the states. In contrast to the standard semantics, messages now include a unique identifier (i.e., a timestamp λ). These identifiers are required to avoid mixing different messages with the same value (and possibly also with the same sender and/or receiver). More details can be found in [13].

The transition rules of the forward reversible semantics can be found in Fig. 4. They are an easy—and conservative—extension of the semantics in Fig. 3 by adding histories to processes. In the histories, we use terms headed by constructors τ , check , send , rec , spawn , and self to record the steps performed by the forward semantics. Note that the auxiliary function matchrec now deals with messages of the form $\{v, \lambda\}$, trivially extending the original function in the standard semantics by ignoring λ when computing the first matching message.

Rollback Debugging Semantics. Now, we introduce a novel *rollback semantics* to undo the actions of a given process. Here, processes in “rollback” mode are annotated using $\lfloor \rfloor_{\Psi}$, where Ψ is a set with the requested rollbacks. In particular,

$$\begin{array}{l}
 (Seq) \quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi \rightarrow \Gamma; \langle p, \tau(\theta, e) : h, (\theta', e'), q \rangle \mid \Pi} \\
 (Send) \quad \frac{\theta, e \xrightarrow{\text{send}(p'', v)} \theta', e' \quad \lambda \text{ is a fresh identifier}}{\Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi \rightarrow \Gamma \cup \{p'', \{v, \lambda\}\}; \langle p, \text{send}(\theta, e, p'', \{v, \lambda\}) : h, (\theta', e'), q \rangle \mid \Pi} \\
 (Receive) \quad \frac{\theta, e \xrightarrow{\text{rec}(\kappa, \overline{cl}_n)} \theta', e' \quad \text{matchrec}(\overline{cl}_n, q) = (\theta_i, e_i, \{v, \lambda\})}{\Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi \rightarrow \Gamma; \langle p, \text{rec}(\theta, e, \{v, \lambda\}, q) : h, (\theta' \theta_i, e' \{ \kappa \mapsto e_i \}), q \setminus \{v, \lambda\} \rangle \mid \Pi} \\
 (Spawn) \quad \frac{\theta, e \xrightarrow{\text{spawn}(\kappa, a/n, [\overline{v}_n])} \theta', e' \quad p' \text{ is a fresh pid}}{\Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi \rightarrow \Gamma; \langle p, \text{spawn}(\theta, e, p') : h, (\theta', e' \{ \kappa \mapsto p' \}), q \rangle \mid \langle p', [], (id, \text{apply } a/n (\overline{v}_n)), [] \rangle \mid \Pi} \\
 (Self) \quad \frac{\theta, e \xrightarrow{\text{self}(\kappa)} \theta', e'}{\Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi \rightarrow \Gamma; \langle p, \text{self}(\theta, e) : h, (\theta', e' \{ \kappa \mapsto p \}), q \rangle \mid \Pi} \\
 (Sched) \quad \frac{}{\Gamma \cup \{p, \{v, \lambda\}\}; \langle p, h, (\theta, e), q \rangle \mid \Pi \rightarrow \Gamma; \langle p, h, (\theta, e), \{v, \lambda\} : q \rangle \mid \Pi}
 \end{array}$$

Fig. 4. Forward reversible semantics

we consider the following rollbacks to undo the actions of a given process in a causal-consistent way:

- **s**: one backward step;
- λ^\uparrow : a backward derivation up to the sending of a message labelled with λ ;
- λ^\downarrow : a backward derivation up to the delivery of a message labelled with λ ;
- λ^{rec} : a backward derivation up to the receive of a message labelled with λ ;
- sp_p : a backward derivation up to the spawning of the process with pid p ;
- **sp**: a backward derivation up to the creation of the annotated process;
- **X**: a backward derivation up to the introduction of variable X .

In the following, in order to simplify the reduction rules, we consider that our semantics satisfies the following *structural equivalence*:

$$\begin{array}{l}
 (SC1) \quad \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_\emptyset \mid \Pi \equiv \Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi \\
 (SC2) \quad \Gamma; \lfloor \langle p, [], (\theta, e), [] \rangle \rfloor_\Psi \mid \Pi \equiv \Gamma; \langle p, [], (\theta, e), [] \rangle \mid \Pi
 \end{array}$$

Therefore, when the set of rollbacks is empty or the process is back to its initial state, we consider that the required rollback has been completed.

Our rollback debugging semantics is modelled with the reduction relation \leftarrow , defined by the rules in Fig. 5. Here, we assume that $\Psi \neq \emptyset$ (but Ψ' might be empty). Let us briefly explain the rules of the rollback semantics:

- Some actions can be directly undone. This is the case dealt with by rules *Seq*, *SendI*, *Receive*, *SpawnI*, *Self*, and *Sched*. In every rule, we remove the corresponding rollback request from Ψ . In particular, all of them remove **s** (since a causal-consistent step has been performed). Rule *Seq* additionally removes the variables whose bindings were introduced in the last step; rule *SendI* removes λ^\uparrow (representing the sending of the message with identifier λ);

$$\begin{array}{l}
\overline{(\text{Seq})} \quad \Gamma; \llbracket \langle p, \tau(\theta, e) : h, (\theta', e'), q \rangle \rrbracket_{\Psi} \mid \Pi \leftarrow \Gamma; \llbracket \langle p, h, (\theta, e), q \rangle \rrbracket_{\Psi \setminus (\{s\} \cup \mathcal{V})} \mid \Pi \\
\text{where } \mathcal{V} = \text{Dom}(\theta') \setminus \text{Dom}(\theta) \\
\overline{(\text{SendI})} \quad \Gamma \cup \{(p', \{v, \lambda\})\}; \llbracket \langle p, \text{send}(\theta, e, p', \{v, \lambda\}) : h, (\theta', e'), q \rangle \rrbracket_{\Psi} \mid \Pi \\
\leftarrow \Gamma; \llbracket \langle p, h, (\theta, e), q \rangle \rrbracket_{\Psi \setminus \{s, \lambda^{\dagger}\}} \mid \Pi \\
\overline{(\text{Send2})} \quad \Gamma; \llbracket \langle p, \text{send}(\theta, e, p', \{v, \lambda\}) : h, (\theta', e'), q \rangle \rrbracket_{\Psi} \mid \llbracket \langle p', h', (\theta'', e''), q' \rangle \rrbracket_{\Psi'} \mid \Pi \\
\leftarrow \Gamma; \llbracket \langle p, \text{send}(\theta, e, p', \{v, \lambda\}) : h, (\theta', e'), q \rangle \rrbracket_{\Psi} \mid \llbracket \langle p', h', (\theta'', e''), q' \rangle \rrbracket_{\Psi' \cup \{\lambda^{\dagger}\}} \mid \Pi \\
\text{if } (p', \{v, \lambda\}) \text{ does not occur in } \Gamma \text{ and } \lambda^{\dagger} \notin \Psi' \\
\overline{(\text{Receive})} \quad \Gamma; \llbracket \langle p, \text{rec}(\theta, e, \{v, \lambda\}, q) : h, (\theta', e'), q \rrbracket_{\Psi \setminus \{v, \lambda\}} \rrbracket_{\Psi} \mid \Pi \\
\leftarrow \Gamma; \llbracket \langle p, h, (\theta, e), q \rangle \rrbracket_{\Psi \setminus \{s, \lambda^{\text{rec}}\}} \mid \Pi \\
\overline{(\text{SpawnI})} \quad \Gamma; \llbracket \langle p, \text{spawn}(\theta, e, p'') : h, (\theta', e'), q \rangle \rrbracket_{\Psi} \mid \llbracket \langle p'', [], (\theta'', e''), [] \rangle \rrbracket_{\Psi'} \mid \Pi \\
\leftarrow \Gamma; \llbracket \langle p, h, (\theta, e), q \rangle \rrbracket_{\Psi \setminus \{s, \text{sp}, p''\}} \mid \Pi \\
\overline{(\text{Spawn2})} \quad \Gamma; \llbracket \langle p, \text{spawn}(\theta, e, p'') : h, (\theta, e), q \rangle \rrbracket_{\Psi} \mid \llbracket \langle p'', h'', (\theta'', e''), q'' \rangle \rrbracket_{\Psi'} \mid \Pi \\
\leftarrow \Gamma; \llbracket \langle p, \text{spawn}(\theta, e, p'') : h, (\theta, e), q \rangle \rrbracket_{\Psi} \mid \llbracket \langle p'', h'', (\theta'', e''), q'' \rangle \rrbracket_{\Psi' \cup \{\text{sp}\}} \mid \Pi \\
\text{if } h'' \neq [] \vee q'' \neq [] \text{ and } \text{sp} \notin \Psi' \\
\overline{(\text{Self})} \quad \Gamma; \llbracket \langle p, \text{self}(\theta, e) : h, (\theta', e'), q \rangle \rrbracket_{\Psi} \mid \Pi \leftarrow \Gamma; \llbracket \langle p, h, (\theta, e), q \rangle \rrbracket_{\Psi \setminus \{s\}} \mid \Pi \\
\overline{(\text{Sched})} \quad \Gamma; \llbracket \langle p, h, (\theta, e), \{v, \lambda\} : q \rangle \rrbracket_{\Psi} \mid \Pi \leftarrow \Gamma \cup \{(p, \{v, \lambda\})\}; \llbracket \langle p, h, (\theta, e), q \rangle \rrbracket_{\Psi \setminus \{s, \lambda^{\dagger}\}} \mid \Pi \\
\text{if the topmost } \text{rec}(\dots) \text{ item in } h \text{ (if any) has the} \\
\text{form } \text{rec}(\theta', e', \{v', \lambda'\}, q') \text{ with } q' \setminus \{v', \lambda'\} \neq \{v, \lambda\} : q
\end{array}$$

Fig. 5. Rollback debugging semantics

rule $\overline{\text{Receive}}$ removes λ^{rec} (representing the receiving of the message with identifier λ); rule $\overline{\text{SpawnI}}$ removes $\text{sp}_{p''}$ (representing the spawning of the process with pid p''); and rule $\overline{\text{Sched}}$ removes λ^{\dagger} (representing the delivery of the message with identifier λ). Note also that rule $\overline{\text{Sched}}$ requires a side condition to avoid the (incorrect) commutation of rules $\overline{\text{Receive}}$ and $\overline{\text{Sched}}$ (see [13] for more details on this issue).

- Other actions require some dependencies to be undone first. This is the case of rules $\overline{\text{Send2}}$ and $\overline{\text{Spawn2}}$. In the first case, rule $\overline{\text{Send2}}$ applies in order to “propagate” the rollback mode to the receiver of the message, so that rules $\overline{\text{Sched}}$ and $\overline{\text{SendI}}$ can be eventually applied. In the second case, rule $\overline{\text{Spawn2}}$ applies to propagate the rollback mode to process p'' so that, eventually, rule $\overline{\text{SpawnI}}$ can be applied. Observe that the rollback sp introduced by the rule $\overline{\text{Spawn2}}$ does not need to be removed from Ψ since the complete process is deleted from Π in rule $\overline{\text{SpawnI}}$.

The correctness of the new rollback semantics can be shown following a similar scheme as in [13] for proving the correctness of the rollback semantics for checkpoints.

We now introduce an operator that performs a causal-consistent backward derivation and is parameterised by a system, a pid and a set of rollback requests:

$$\text{rb}(\Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi, p, \Psi) = \Gamma'; \Pi' \quad \text{if } \Gamma; \llbracket \langle p, h, (\theta, e), q \rangle \rrbracket_{\Psi} \mid \Pi \leftarrow^* \Gamma'; \Pi' \not\leftarrow$$

The operator adds a set of rollback requests to a given process² and then performs as many steps as possible using the rollback debugging semantics.

By using the above parametric operator, we can easily define several rollback operators that are useful for debugging. Our first operator, $\text{rollback}(\Gamma; \Pi, p)$, just performs a causal-consistent backward step for process p :

$$\text{rollback}(\Gamma; \Pi, p) = \text{rb}(\Gamma; \Pi, p, \{s\})$$

Notice that this may trigger the execution of any number of backward steps in other processes in order to first undo the consequences, if any, of the step in p .

This operator can easily be extended to an arbitrary number of steps:

$$\text{rollback}(\Gamma; \Pi, p, n) = \begin{cases} \Gamma; \Pi & \text{if } n = 0 \\ \text{rollback}(\Gamma'; \Pi', p, n - 1) & \text{if } n > 0 \text{ and} \\ \text{rollback}(\Gamma; \Pi, p) = \Gamma'; \Pi' & \end{cases}$$

Also, we might be interested in going backward until a relevant action is undone. For instance, we introduce below operators that go backward up to, respectively, the sending of a message with a particular identifier λ , the receiving of a message with a particular identifier λ , and the spawning of a process with pid p' :

$$\begin{aligned} \text{rollback}(\Gamma; \Pi, p, \lambda^\uparrow) &= \text{rb}(\Gamma; \Pi, p, \{\lambda^\uparrow\}) \\ \text{rollback}(\Gamma; \Pi, p, \lambda^{\text{rec}}) &= \text{rb}(\Gamma; \Pi, p, \{\lambda^{\text{rec}}\}) \\ \text{rollback}(\Gamma; \Pi, p, \text{sp}_{p'}) &= \text{rb}(\Gamma; \Pi, p, \{\text{sp}_{p'}\}) \end{aligned}$$

Note that p is a parameter of the three operators, but it could also be automatically computed (from λ in the first two rules, from p' in the last one) by inspecting the histories of the processes in Π . This is actually what CauDER does.

Finally, we consider an operator that performs backward steps up to the introduction of a binding for a given variable:

$$\text{rollback}(\Gamma; \Pi, p, X) = \text{rb}(\Gamma; \Pi, p, \{X\})$$

Here, p cannot be computed automatically from X , since variables are local and, hence, variable X may occur in several processes; thus, p is needed to uniquely identify the process of interest.³

4 CauDER: A Causal-Consistent Reversible Debugger

The CauDER implementation is conveniently bundled together with a graphical user interface to facilitate the interaction of users with the reversible debugger.

CauDER works as follows: when it is started, the first step is to select an Erlang source file. The selected source file is then translated into Core Erlang, and the

² Actually, in this work, we only consider a single rollback request at a time, so Ψ is always a singleton. Nevertheless, our formalisation considers that Ψ is a set for notational convenience and, also, in order to accept multiple rollbacks in the future.

³ Actually, in CauDER, uniqueness of variable names is enforced via renaming.

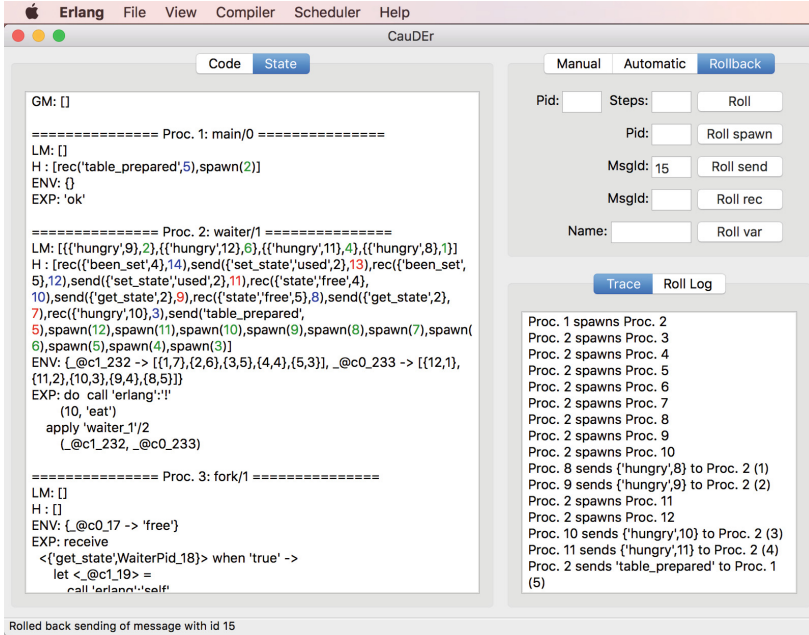


Fig. 6. CauDER screenshot

resulting code is shown in the Code tab. Then, the user can choose any of the functions from the module and write the arguments that she wants to evaluate the function with. An initial system state, with an empty global mailbox and a single process performing the specified function application, appears in the State tab when the user presses the START button. Now, the user can explore possible program executions both forward and backward, according to three different modes, corresponding to the three tabs on the top right of the window in Fig. 6. In the Manual mode, the user selects a process or message identifier, and buttons corresponding to forward and backward enabled reductions for the chosen process/message are available. Note that a backward reduction is *enabled* only if the action has no causal dependencies that need to be undone (single backward reductions correspond to applications of rules *Seq*, *Send1*, *Receive*, *Spawn1*, *Self*, and *Sched* in Fig. 5, see the uncontrolled reversible semantics in [13] for more details). In the Automatic mode one can decide the direction (forward or backward) and the number of steps to be performed. Actual steps are selected by a suitable scheduler. Currently, two (random) schedulers are available, one of which gives priority to processes w.r.t. the scheduling of messages (as in the “normalisation” strategy described in [13]), while the other has a uniform distribution. None of these schedulers mimics the Erlang/OTP scheduler. Indeed, it would be very hard to replicate this behaviour, as it depends on many parameters (threads, workload, etc.). However, this is not necessary, since we are only interested in reproducing the errors that occur in actual executions, and we discuss in

future work how to obtain this without the need of mimicking the Erlang/OTP scheduler. The **Automatic** tab also includes a **Normalize** button, that executes all enabled actions but message schedulings. The last tab, **Rollback**, implements the rollback operators described in Sect. 3.

While exploring the execution, two tabs are updated to provide information on the system and its execution. The **State** tab describes the current system, including the global mailbox **GM**, and, for each process, the following components: the local mailbox **LM**, the history **H**, the environment **ENV**, and the expression under evaluation **EXP**. Identifiers of messages are highlighted in colour. This tab can be configured to hide any component of the process representation. Also, we consider two levels of abstraction for both histories and environments: for histories, we can either show all the actions or just the concurrent actions (send, receive and spawn); for environments, we can either show all variable bindings (called the *full* environment) or only the bindings for those variables occurring in the current expression (called the *relevant* environment).

The **Trace** tab gives a linearised description of the concurrent actions performed in the system, namely sends and receives of messages, and spawns of processes. This is aimed at giving a global picture of the system evolution, to highlight anomalies that might be caused by bugs.

A further tab is available, **Roll Log**, which is updated in case of rollbacks. It shows which actions have been actually undone upon a rollback request. This tab allows one to understand the causal dependencies of the target process of the rollback request, frequently highlighting undesired or missing dependencies directly caused by bugs.

The release version (v1.0) of CauDEr is fully written in Erlang, and it is publicly available from <https://github.com/mistupv/cauder> under the MIT license. The only requirement to build the application is to have Erlang/OTP installed and built with wxWidgets. The repository also includes some documentation and a few examples to easily test the application.

4.1 The CauDEr Workflow

A typical debugging session with CauDEr proceeds as follows. First, the user may run the program some steps forward using the **Automatic** mode in order to exercise the code. After each sequence of forward steps, she looks at the program output (which is not on the CauDEr window, but in the console where CauDEr has been launched) and possibly at the **State** and **Trace** tabs to check for abnormal behaviours. The **State** tab helps to identify these behaviours within a single process, while the **Trace** tab highlights anomalies in the global behaviour.

If the user identifies an unexpected action, she can undo it by using any (or a combination) of the available rollback commands. The **Roll Log** tab provides information on the causal-consistent rollbacks performed (in some cases, this log is enough to highlight the bug). From there, the user typically switches to the **Manual** mode in order to precisely control the doing or undoing of actions in a specific state. This may involve performing other rollbacks to reach previous

states. Our experience says that inspecting the full environment during the Manual exploration is quite helpful to locate bugs caused by sequential code.

4.2 Finding Concurrency Bugs with CauDER

We use as a running example to illustrate the use of our debugger the well-known problem of dining philosophers. Here, we have a process for each philosopher and for each fork. We avoid implementations that are known to deadlock by using an arbitrator process, the waiter, that acts as an intermediary between philosophers and forks. In particular, if a philosopher wants to eat, he asks the waiter to get the forks. The waiter checks whether both forks are free or not. In the first case, he asks the forks to become *used*, and sends a message *eat* to the philosopher. Otherwise he sends a message *think* to the philosopher. When a philosopher is done eating, he sends a message *eaten* to the waiter, who in turn will release (i.e., set to *free*) the corresponding forks. The full Erlang code of the (correct) example, `dining.erl`, is available from <https://github.com/mistupv/dining-philos>.

Message Order Violation Scenario. Here, we consider the buggy version of the program that can be found in file `dining_simple_bug.erl` of the above repository. In this example, running the program forward using the Automatic mode for about 600 steps is enough to discern something wrong. In particular, the user notices in the output that some philosophers are told to think when they should be told to eat, even at the beginning of the execution. Since the bug appears so early, it is probably a local bug, hence the user first focuses on the State tab. When the user considers the waiter process, she sees in the history an unexpected sequence of concurrent events of the following form (shown in reverse chronological order):

```
... ,send('think',10),rec('free',9),send({'get_state',2},8),
rec({'hungry',12},6),send({'get_state',2},7),rec({'hungry',9},2), ...
```

Here, the waiter has requested the state of a fork with `send({'get_state',2},7)`, where 2 is the process id of the waiter itself and 7 the message id. Unexpectedly, the waiter has received a message *hungry* as a reply, instead of a message *free* or *used*. To get more insight on this, the user decides to rollback the receive of `{'hungry',12}`, which has 6 as message id. As a result, the rollback gets the system back to a state where `send({'get_state',2},7)` is the last concurrent event for the waiter process. Finally, the user switches to the Manual mode and notices that the next available action for the waiter process is to receive the message `{'hungry',12}` in the receive construct from the `ask_state` function. Function `ask_state` is called by the waiter process when it receives a *hungry* request from a philosopher (to get the state of the two forks). Obviously, a further message *hungry* should not be received here. The user easily realises then that the pattern in the receive is too general (in fact, it acts as a catch-all clause) and, as a result, the receive is matching also messages from other forks and even philosophers. Indeed, after

sending the message `get_state` to a fork, the programmer assumed that the next incoming message will be the state of the fork. However, the function is being evaluated in the context of the waiter process, where many other messages could arrive, e.g., messages `hungry` or `eaten` from philosophers.

It would not be easy to find the same bug using a standard debugger. Indeed, one would need to find where the wrong message `hungry` is sent, and put there a breakpoint. However, in many cases, no scheduling error will occur, hence many attempts would be needed. With a standard reversible debugger (like Actoverse [19]) one could look for the point where the wrong message is received, but it would be difficult to stop the execution at the exact message. Watch points do not help much, since all such messages are equal, but only some of them are received in the wrong `receive` operation. Indeed, in this example, the CauDER facility of rollbacking a specific message receiving, coupled with the addition of unique identifiers to messages, is a key in ensuring the success of the debugging session.

Livelock Scenario. Now, we consider the buggy version of the dining philosophers that can be found in file `dining_bug.erl` of our repository. In this case, the output of the program shows that, after executing some 2000 steps with the Automatic mode, some philosophers are always told to think, while others are always told to eat. In contrast to the previous example, this bug becomes visible only late in the execution, possibly only after some particular pattern of message exchanges has taken place (this is why it is harder to debug). In order to analyse the message exchanges the user should focus on the Trace tab first. By carefully examining it, the user realises that, in some cases, after receiving a message `eaten` from a philosopher, the waiter sends the two messages `{'set_state','free',2}` to release the forks to the same fork:

```
Proc. 2 receives {'eaten',10} (28)
Proc. 2 sends {'set_state','free',2} to Proc. 5 (57)
Proc. 5 receives {'set_state','free',2} (57)
Proc. 5 sends {'been_set',5} to Proc. 2 (58)
Proc. 2 receives {'been_set',5} (58)
Proc. 2 sends {'set_state','free',2} to Proc. 5 (59)
Proc. 5 receives {'set_state','free',2} (59)
Proc. 5 sends {'been_set',5} to Proc. 2 (60)
Proc. 2 receives {'been_set',5} (60)
```

Then, the user rollback the sending of the last message from the waiter process (the one with message id 59) and chooses to show the full environment (a clever decision). Surprisingly, the computed values for `LeftForkId` and `RightForkId` are equal. She decides to rollback also the sending of message with id 57, but she cannot see anything wrong there, so the computed value for `RightForkId` must be wrong. Now the user focuses on the corresponding line on the code, and she notices that the operands of the modulo operator have been swapped, which is the source of the erroneous behaviour.

This kind of livelocks are typically hard to find with other debugging tools. For instance, `Concuerror` [10] requires a finite computation, which is not the case in this scenario where the involved processes keep doing actions all the time but no global progress is achieved (i.e., some philosophers never eat).

5 Related Work

Causal-consistent debugging has been introduced by `CaReDeb` [7], in the context of language μOz . The present paper improves on `CaReDeb` in many directions. First, μOz is only a toy language where no realistic programs can be written (e.g., it supports only integers and a few arithmetic operations). Second, μOz is not distributed, since messages are atomically moved from the sender to a message queue, and from the queue to the target process. This makes its causality model, hence the definition of a causal-consistent reversible semantics, much simpler. Third, in [7] the precise semantics of debugging operators is not fully specified. Finally, the implementation described in [7] is just a proof-of-concept.

More in general, our work is in the research thread of causal-consistent reversibility (see [12] for a survey), first introduced in [4] in the context of process calculus CCS. Most of the works in this area are indeed on process calculi, but for the work on μOz already discussed (the theory was introduced in [14]) and a line of work on the coordination language μkclaim [8]. However, μkclaim is a toy language too. Hence, we are the first ones to consider a mainstream programming language. A first approach to the definition of a causal-consistent semantics of Erlang was presented in [17], and extended in [13]. While we based `CauDER` on the uncontrolled semantics therein (and on its proof-of-concept implementation), we provided in the present paper an updated controlled semantics more suitable for debugging, and a mature implementation with a complete interface and many facilities for debugging. Moreover, our tool is able to deal with a larger subset of the language, mainly in terms of built-in functions and data structures.

While `CaReDeb` is the only other causal-consistent debugger we are aware of, two other reversible debuggers for actor systems exist. `Actoverse` [19] deals with Akka-based applications. It provides many relevant features which are complementary to ours. These include a partial-order graphical representation of message exchanges that would nicely match our causal-consistent approach, message-oriented breakpoints that allow one to force specific interleavings in message schedulings, and facilities for session replay to ensure bugs reappear when executing forward again. In contrast, `Actoverse` provides less facilities for state inspection and management than us (e.g., it has nothing similar to our `Roll var` command). Also, the paper does not include any theoretical framework defining the behaviour of the debugger. `EDD` is a declarative debugger for Erlang (see [1] for a version dealing with sequential Erlang). `EDD` tracks the concurrent actions of an execution and allows the user to select any of them to start the questions. Declarative debugging is essentially orthogonal to our approach.

`Causeway` [20] is not a full-fledged debugger but a post-mortem trace analyser, i.e., it performs no execution, but just explores a trace of a run. It concentrates on message passing aspects, e.g., it does not allow one to explore the

state of single processes (states are not in the logs analysed by Causeway). On the contrary it provides nice mechanisms to abstract and filter different kinds of communications, allowing the user to decide at each stage of the debugging process which messages are of interest. These mechanisms would be an interesting addition for CauDEr.

6 Discussion

In this work, we have presented the design of CauDEr, a causal-consistent reversible debugger for Erlang. It is based on the reversible semantics introduced in [13, 17], though we have introduced in this paper a new rollback semantics which is especially appropriate for debugging Erlang programs. We have shown in the paper that some bugs can be more easily located using our new tool, thus filling a gap in the collection of debugging tools for Erlang.

Currently, our debugger may run a program either forward or backward (in the latter case, in a causal-consistent way). After a backward computation that undoes some steps, we can resume the forward computation, though there are no guarantees that we will reproduce the previous forward steps. Some debuggers (so-called omniscient or back-in-time debuggers) allow us to move both forward and backward along a *particular* execution. As a future work, we plan to define a similar approach but ensuring that once we resume a forward computation, we can follow the same previous forward steps *or some other causal-consistent steps*. Such an approach might be useful, e.g., to determine which processes depend on a particular computation step and, thus, ease the location of a bug.

Another interesting line of future work involves the possibility of capturing a faulty behaviour during execution in the standard environment, and then replaying it in the debugger. For instance, we could instrument source programs so that their execution in a standard environment writes a log in a file. Then, when the program ends up with an error, we could use this log as an input to the debugger in order to explore this particular faulty behaviour (as postmortem debuggers do). This approach can be applied even if the standard environment is distributed and there is no common notion of time, since causal-consistent reversibility relies only on a notion of causality.

For the same reason we could also develop a fully distributed debugger, where each process is equipped with debugging facilities, and a central console allows us to coordinate them. This would strongly improve scalability, since most of the computational effort (running and backtracking programs) would be distributed. However, this step requires a semantics without any synchronous interaction (e.g., rules $\overline{Send2}$ and $\overline{Spawn2}$ would need to be replaced by a more complex asynchronous protocol).

Acknowledgements. The authors gratefully acknowledge the anonymous referees for their useful comments and suggestions.

References

1. Caballero, R., Martin-Martin, E., Riesco, A., Tamarit, S.: EDD: a declarative debugger for sequential Erlang programs. In: *Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 581–586. Springer, Heidelberg (2014).* https://doi.org/10.1007/978-3-642-54862-8_49
2. Carlsson, R., et al.: Core Erlang 1.0.3 language specification (2004). <https://www.it.uu.se/research/group/hipecerl/doc/core.erlang-1.0.3.pdf>
3. Claessen, K., et al.: Finding race conditions in Erlang with QuickCheck and PULSE. In: *ICFP, pp. 149–160. ACM (2009)*
4. Danos, V., Krivine, J.: Reversible communicating systems. In: *Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004).* https://doi.org/10.1007/978-3-540-28644-8_19
5. D’Osualdo, E., Kochems, J., Ong, C.-H.L.: Automatic verification of Erlang-style concurrency. In: *Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 454–476. Springer, Heidelberg (2013).* https://doi.org/10.1007/978-3-642-38856-9_24
6. Fredlund, L.A., Svensson, H.: McErlang: a model checker for a distributed functional programming language. In: *ICFP, pp. 125–136. ACM (2007)*
7. Giachino, E., Lanese, I., Mezzina, C.A.: Causal-consistent reversible debugging. In: *Gnesi, S., Rensink, A. (eds.) FASE 2014. LNCS, vol. 8411, pp. 370–384. Springer, Heidelberg (2014).* https://doi.org/10.1007/978-3-642-54804-8_26
8. Giachino, E., Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent rollback in a tuple-based language. *J. Log. Algebr. Meth. Program.* **88**, 99–120 (2017)
9. Giantsios, A., Papaspyrou, N., Sagonas, K.: Concolic testing for functional languages. In: *PPDP, pp. 137–148. ACM (2015)*
10. Gotovos, A., Christakis, M., Sagonas, K.: Test-driven development of concurrent programs using Concuerror. In: *10th ACM SIGPLAN Workshop on Erlang, pp. 51–61. ACM (2011)*
11. Landauer, R.: Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.* **5**, 183–191 (1961)
12. Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent reversibility. *Bull. EATCS* **114**, 19 (2014)
13. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang (2017). Submitted for publication. <http://users.dsic.upv.es/~gvidal/lmpv17/paper.pdf>
14. Lienhardt, M., Lanese, I., Mezzina, C.A., Stefani, J.-B.: A reversible abstract machine and its space overhead. In: *Giese, H., Rosu, G. (eds.) FMOODS/FORTE -2012. LNCS, vol. 7273, pp. 1–17. Springer, Heidelberg (2012).* https://doi.org/10.1007/978-3-642-30793-5_1
15. Lindahl, T., Sagonas, K.: Practical type inference based on success typings. In: *PPDP, pp. 167–178. ACM Press (2006)*
16. Lopez, C.T., Marr, S., Mössenböck, H., Boix, E.G.: A study of concurrency bugs and advanced development support for actor-based programs. *CoRR abs/1706.07372* (2017)
17. Nishida, N., Palacios, A., Vidal, G.: A reversible semantics for Erlang. In: *Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 259–274. Springer, Cham (2017).* https://doi.org/10.1007/978-3-319-63139-4_15
18. Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: *10th ACM SIGPLAN Workshop on Erlang, pp. 39–50. ACM (2011)*

19. Shibantai, K., Watanabe, T.: Actoverse: a reversible debugger for actors. In: AGERE, pp. 50–57. ACM (2017)
20. Stanley, T., Close, T., Miller, M.S.: Causeway: a message-oriented distributed debugger. Technical report, HPL-2009-78 (2009). <http://www.hpl.hp.com/techreports/2009/HPL-2009-78.html>
21. Svensson, H., Fredlund, L.A., Earle, C.B.: A unified semantics for future Erlang. In: 9th ACM SIGPLAN Workshop on Erlang, pp. 23–32. ACM (2010)



Cheap Remarks About Concurrent Programs

Michael Walker^(✉) and Colin Runciman

University of York, York, UK
{m5w504,colin.runciman}@york.ac.uk

Abstract. We present CoCo, the Concurrency Commentator, a tool that recovers a declarative view of concurrent Haskell functions operating on some shared state. This declarative view is presented as a collection of automatically discovered properties. These properties are about *refinement and equivalence of effects*, rather than equality of final results. The tool is based on testing in a dynamically pruned search-space, rather than static analysis or theorem proving. Case studies about concurrent stacks and semaphores demonstrate how use of CoCo can inform understanding of program behaviour.

1 Introduction

Concurrency is a necessary paradigm for many applications, and yet it is difficult to get right in an imperative setting, where the order of effects is both important and unpredictable. Declarative programming, whether logical or functional, offers the promise of a simpler alternative; the programmer describes the desired program, and does not need to worry about the explicit order of effects.

Haskell is a purely functional language. Concurrency in Haskell is modelled with a monad abstraction which is built on top of effectful operations on shared state [11]. Once again, the order of effects is both important and unpredictable. Concurrent Haskell programs are not so declarative.

In this paper we present CoCo, a tool to recover a declarative view of concurrent programs. CoCo takes as input a collection of operations on some shared mutable state. CoCo outputs are declarative properties of equivalence and refinement between concurrent expressions: see Sects. 3 and 5 for examples.

A list of such declarative properties is useful in a number of ways: (1) it can be an addition to existing documentation; (2) the programmer may gain new insights about their program; and (3) the absence of expected properties, or the presence of unexpected ones, may indicate an error.

Our technique uses testing, so is potentially unsound. A property is a conjecture supported only by a finite set of test cases. So some reported properties may not hold in general.

Contributions. We present a method, based on program synthesis and systematic concurrency testing, to discover properties of stateful operations in a declarative

language. Furthermore, we demonstrate the viability of this method by implementing the CoCo tool in Haskell. We then obtain illustrative results from CoCo for some simple applications.

Roadmap. The rest of the paper is structured as follows: Sect. 2 introduces three key concerns in the implementation of a tool such as CoCo. Section 3 gives an introductory example. Section 4 gives a detailed discussion of how CoCo generates terms and discovers properties. Section 5 presents two case studies. Section 6 considers related work and how our contributions differ from it. Section 7 presents conclusions and evaluates the approach.

2 Key Concerns of Observing Concurrent Programs

In implementing a tool to discover properties of *concurrent* programs, we have some concerns which are not applicable to sequential programs. Firstly, concurrent programs are nondeterministic; so if we simply compared results of single executions, the discovered properties may not hold in the general case. Secondly, mutable state is subject to interference from other threads; so if we do not consider concurrent interference, the discovered properties may not hold when there are more threads involved. Finally, we need to decide what it means for two concurrent programs to be related.

Nondeterminism. If we restrict the nondeterminism in our program to schedule nondeterminism, we can use systematic concurrency testing (SCT) [4, 7, 9, 10] techniques. These techniques aim to test a variety of schedules, making use of runtime knowledge of the program to reduce the number of required executions, without necessarily sacrificing completeness.

We have previously developed Déjà Fu [14] an SCT tool for Haskell, based on a typeclass-abstraction over the primitive concurrency operations. CoCo uses Déjà Fu to produce the set of results of a generated program fragment.

Interference. We do not know what sort of interference may lead to interesting results. So CoCo requires the programmer to supply a function with effects, which is executed concurrently during property discovery, to provide this interference. By supplying different sorts of interference, the programmer can see how the API they provide behaves in different concurrent contexts.

Properties. We formulate our properties in terms of *observational refinement* [8], where the observations we take are snapshots of the shared state. CoCo requires the programmer to supply an observation function to produce these snapshots. By varying their observation function, the programmer can see different aspects of the API they provide.

We define a *behaviour* of a concurrent program as a pair of a final observation, taken after the program terminates, and a possible *failure*. Failures are states like a deadlock, or an uncaught exception. By considering the set of a program's


```

type C = Concurrency

sig :: Sig (MVar C Int) (Maybe Int) (Maybe Int)
sig = Sig
  { initialise = maybe newEmptyMVar newMVar
  , expressions =
    [ -- example 1
      lit "putMVar" (putMVar :: MVar C Int -> Int -> C ())
    , lit "takeMVar" (takeMVar :: MVar C Int -> C Int)
    , lit "readMVar" (readMVar :: MVar C Int -> C Int)
    ]
  , backgroundExpressions =
    [ -- example 2
      lit "tryPutMVar" (tryPutMVar :: MVar C Int -> Int -> C Bool)
    ]
  , interfere = \v _ -> putMVar v 42
  , observe = \v _ -> tryReadMVar v
  , backToSeed = \v _ -> tryReadMVar v
  }

```

Fig. 1. CoCo signature for MVars holding Ints.

possible behaviours, rather than simply final observations, we can distinguish between operations which may fail and those which do not. Properties that we report are of the form $A \text{ === } B$, meaning that the sets of behaviours of A and B are equal; and $A \text{ ->- } B$, meaning that the set of behaviours of A is a strict subset of the set of behaviours of B .

3 An Illustrative Example

Let us now show an example use of CoCo. We consider a type of concurrent shared variable in the Haskell libraries. An `MVar` is a mutable memory cell which may be *full* or *empty*. Instead of the standard version of the functions from Haskell's `Control.Concurrent` library module, we instead use typeclass-generalised versions which Déjà Fu can test. We shall examine three basic operations over `MVars`: `put`, `take`, and `read`. To *put* is to block until the `MVar` is empty and then set its value. To *take* is to block until the `MVar` is full, remove its value, and return the value. To *read* is to *take*, but without emptying the `MVar`. Each function has a non-blocking *try* variant, which returns an indicator of success.

Allowing shared values of type `Int`, we have the following type signatures:

```

putMVar  :: MVar Concurrency Int -> Int -> Concurrency ()
takeMVar :: MVar Concurrency Int -> Concurrency Int
readMVar :: MVar Concurrency Int -> Concurrency Int

```

Here `Concurrency` is an implementation of the concurrency-typeclass. In this case, the return type of each function is of the form `Concurrency x`, meaning

that the result of the function produces an `x` value and also has some effects in the concurrency execution context. The `MVar` type is parameterised by the monad type; `MVar` is an abstract type, with the concrete type determined by the monad.

Table 1. The behaviours of the terms in property (2).

Term	Seed	Final state	Deadlocks
Read	Nothing	Just 42	No
	Just 0	Just 0	No
Take/Put	Nothing	Just 42	No
	Just 0	Just 0	No
		Just 42	Yes

Signatures. When we use `CoCo`, we must provide the functions and values which may appear in properties. We must also provide a way to initialise the state, an observation function, and an interference function. We call this collection of programmer-supplied definitions the *signature*.

Figure 1 shows a signature for `MVar` operations. The initialisation function constructs an empty or a full `MVar`. The interference function simply stores a new value. The observation function takes a snapshot of the state. The `backToSeed` function is used to check whether the state has been changed: if the original and final seed values are the same, the state is unchanged.

It is essential to provide an initialisation function which gives a representative collection of states, and an interference function which can disrupt the functions of interest. If our initialisation function only produced a full `MVar`, we could find properties which do not hold when the `MVar` is empty. Because our interference function only writes to the `MVar`, we may find properties which do not hold when there are multiple consumers. Developing a fuller understanding of the functions under test may require examining the different property-sets found under different execution conditions.

MVar properties. Given `putMVar`, `takeMVar`, and `readMVar`, `CoCo` produces:

- `readMVar @ === readMVar @ >> readMVar @` (1)
- `readMVar @ ->- takeMVar @ >>= \x -> putMVar @ x` (2)
- `takeMVar @ === readMVar @ >> takeMVar @` (3)
- `putMVar @ x === putMVar @ x >> readMVar @` (4)

Here `@` is the state argument, in this case the `MVar`.

Property (1) shows that `readMVar` is idempotent; (2) shows that it is not merely a take followed by a put, it is rather a distinct operation; (3) and (4) show that it does not modify the `MVar`, and that it does not block when the `MVar` is full. Property (4) may appear to be type-incorrect, but remember that `CoCo` does not consider equality of term results, only the effects.

We see the effect of the interference in (2): with no other producers, this would be an equivalence; it is only when interference by another thread is introduced that the equivalence breaks down and the distinction is revealed. Table 1 shows the possible behaviours. This property is a strict refinement because, while the behaviours for the seed value `Nothing` are the same, the behaviours of the left term for the seed value `Just 0` are a strict subset of the behaviours of the right.

Background Expressions. Sometimes when expressing properties it is necessary to call upon other expressions which are of secondary interest. Such expressions are commonly called *background* expressions. A property is only reported if each side includes at least one non-background expression. If we include `tryPutMVar` as a background expression, CoCo discovers these additional properties:

```
readMVar @   ===  readMVar @ >> tryPutMVar @ x
readMVar @   ===  readMVar @ >>= \x -> tryPutMVar @ x      (5)
readMVar @   ->-  takeMVar @ >>= \x -> tryPutMVar @ x
putMVar @ x   ===  putMVar @ x >> tryPutMVar @ x1
```

Property (5) shows how important the choice of interference function is. The left and right terms are not equivalent. If the interference were to empty a full `MVar` then the right term could restore its original value. As our interference function only produces, rather than consumes, it will never alter the value in a full `MVar`.

The above example takes about 4s to run in total, and the output displayed here is the output of the tool, aside from the property numbers.

4 How CoCo Works

A simplified version of our approach is to generate all terms up to some syntactic size limit, compute and store their behaviours, and then find properties by comparing the sets of behaviours of each pair of terms. This would be slow, however. Following the lead of QuickSpec [3, 12] we make three key improvements:

1. We generate *schemas* with *holes*, rather than *terms* with *variables* (Sect. 4.1)
2. We only compute the set of behaviours of the most general term of every schema (Sect. 4.2)
3. We interleave property discovery with schema generation, and aggressively prune redundant schemas (Sect. 4.3)

The main difference between our approach and QuickSpec is how we handle monadic operations, and that QuickSpec compares *equality* of term *results* whereas we compare *refinement* of term *behaviours*. Furthermore, we generate lambda-terms in a restricted setting whereas QuickSpec does not do so at all.

4.1 Representing and Generating Expression Schemas

We can greatly reduce the number of expressions by not generating alpha-equivalent ones. Instead of generating an expression like `push @ x >> push @ y` we will instead generate the expression `push @ ? >> push @ ?` where each `?` is a *hole* for a variable. These expressions-with-holes are called *schemas*. One schema can be instantiated into many *terms* by assigning variable names to groups of holes. The push-push schema has two semantically distinct term instances: the single-variable and the two-variable cases.

```

data Expr s h = Lit   String Dynamic
              | Var   TypeRep (Var h)
              | Bind  TypeRep (Expr s h) (Expr s h)
              | Ap    TypeRep (Expr s h) (Expr s h)
              | State

data Var h = Hole h | Named String | Bound Int

type Schema s = Expr s ()
type Term   s = Expr s Void

```

Fig. 2. Representation of Haskell expressions.

Figure 2 shows our expression representation. The `Expr` type is parameterised by the state type and a *hole* type. The state parameter ensures expressions that assume different execution contexts cannot be inadvertently combined. The hole parameter allows for a statically enforced distinction between schemas and terms. Each `Expr` constructor carries around a type (except the state, which is implicit). We hide the details of this representation and provide *smart constructor* functions to ensure only well-typed expressions can be constructed.

Schema Generation. Generating new schemas is straightforward. We give expressions a notion of *size* and generate schemas in size order. The needed expressions of size 1 are supplied in the user’s signature. For larger sizes we combine pairs of appropriately sized known schemas and keep the type-correct ones.

We interleave generation with evaluation and property discovery. In this way we can partition schemas into equivalence classes and use only the smallest of known-equivalent schemas when generating new ones.

Monadic Expressions. The expressions of most interest to us are *monadic* expressions. Such expressions allow us to combine smaller effects to create larger ones. We simplify this task by taking inspiration from Haskell’s `do`-notation. `Do`-notation is a syntactic sugar for expressing sequences of monadic operations in an imperative style, which has explicit variable bindings and makes the sequencing of effects clear. Rather than generating lambda-terms, we use a kind of first-class

do-notation where the monadic bind operation binds the result of evaluating the *binder* to zero or more holes in the *body*. Restricting ourselves to this simpler case allows us to avoid many of the complexities of trying to generate lambda-terms directly.

For example, the schema `pop @ >>= \x -> push @ x` is generated like so:

1. Combine `pop` and `@` to produce `pop @`
2. Combine `push` and `@` to produce `push @`
3. Combine `push @` and `?` to produce `push @ ?`
4. Combine `pop @` and `push @ ?` to produce both `pop @ >> push @ ?` and `pop @ >>= \x -> push @ x`.

Bound variables use de Bruijn indices [5]. Names are only assigned when expressions are displayed to the user.

4.2 Evaluating Most General Terms

Time spent evaluating terms dominates the execution cost of CoCo. In the worst case the number of executions needed for a term is exponential in the number of threads, pre-emptive context switches, and blocking operations [9].

What is more, our term evaluation always involves at least two threads: the term thread executing the term itself, and an *interference thread*. The term thread may fork additional threads. The interference thread is essential to distinguish refinement from equality in some cases, such as in property (2).

To avoid repeated work, we compute the behaviours of all the terms for a schema when it is generated. We annotate each schema with some metadata, including its behaviour-sets, and compare these cached behaviours later when discovering properties. While possibly a significant space cost, storing this data reduces the execution time of some of our test applications from hours to minutes.

Deriving Terms from Schemas. One schema may have many term instances. For example, given a schema with two holes of two types, we can produce four semantically distinct terms, here ordered from most general to most constrained:

```
f (? :: Int) (? :: Bool) (? :: Bool) (? :: Int)
```

```
f (w :: Int) (x :: Bool) (y :: Bool) (z :: Int)
```

```
f (w :: Int) (x :: Bool) (y :: Bool) (w :: Int)
```

```
f (w :: Int) (x :: Bool) (x :: Bool) (z :: Int)
```

```
f (w :: Int) (x :: Bool) (x :: Bool) (w :: Int)
```

We use a simple reduce-and-conquer algorithm to eliminate holes:

1. Pick a type and find the set of all holes of that type.
2. For each partition of the hole-set make a distinct copy of the schema and in each case assign to each subset in the partition a distinct variable name.
3. If there are remaining hole types, continue recursively from (1).

Evaluating Terms. To compute the behaviours of every term for a schema, we need only consider the most general term. The behaviours of all less-general terms can be derived from the most general case by restricting to cases where the variables are equal. For example, given the behaviours of $f\ x\ y$, we throw away those where $x \neq y$ to obtain the behaviours of $f\ x\ x$.

Déjà Fu allows us to make an observation of the final state even if evaluation of the term deadlocks. This is essential, as an operation which deadlocks may have altered the state before blocking.

4.3 Property Discovery and Schema Pruning

Not only do we interleave generation with evaluation, we also interleave it with property-discovery. After all schemas of a given size are generated and their most general terms evaluated, we compare each such new schema against all smaller ones to discover equivalences and refinements.

As one schema may correspond to many terms, we may discover many properties between a pair of schemas. In practice, most of these properties are consequences of more general ones. We solve this problem by first producing all properties between the pair of schemas, and then pruning properties which are simple consequences of another. Property P_2 is made redundant by property P_1 if (1) both P_1 and P_2 are equivalences or both are refinements; and (2) P_1 has a more general allocation of variables to holes. As ->- is *strict* refinement, it is impossible for both $S \text{ === } T$ and $S \text{ ->- } T$ to hold.

Smallest Schemas. To avoid discovering the same property multiple times, we maintain a set of *smallest schemas*. At first we assume all schemas to be smallest. If a syntactically smaller schema is a refinement of a larger one, the larger is annotated as “not smallest”. When generating new monadic binds:

- A schema $S \gg T$ is only generated if both S and T are smallest schemas.
- A schema $S \gg= \lambda x \rightarrow T[x]$ is only generated if T is a smallest schema.

We also only consider properties $S \text{ === } T$ or $S \text{ ->- } T$ where both S and T are smallest schemas.

Neutral Schemas. A schema N is neutral if and only if, for all other schemas S , these identities hold: $N \gg S \text{ === } S \text{ === } S \gg N$. For example, `readMVar` is not a neutral `MVar` operation, as it may block, but the non-blocking alternative `tryReadMVar` is neutral. A sufficient condition for a schema to be neutral is if its most general term instance is (1) always atomic; (2) never fails; and (3) never modifies the state.

We use a heuristic method based on execution traces to determine if a schema is atomic, and use the seed values to determine if it modifies the state. If a schema is judged to be neutral, we do not use it when constructing larger schemas.

Projection to a Common Namespace. We compute the behaviours of every term individually, yet we construct properties from pairs of terms. Each term introduces its own variable namespace: the variable “x” in one term is unrelated to the variable “x” in another. When discovering properties, we must first project both terms into a common namespace. Each variable in each term can either be given a unique name, or identified with a variable in the other term. We never reduce the number of distinct variables in a term. To do so would only reproduce another term generated from the same schema.

As a pair of terms may have many projections, we may discover many properties between them: at most one for each projection. In practice, most of these properties are consequences of more general ones. We only keep the most general.

```

newtype LockStack m a = LockStack (MVar m [a])

push :: MonadConc m => a -> LockStack m a -> m ()
push a (LockStack v) = modifyMVar v (\as -> return (a:as, ()))

pop :: MonadConc m => LockStack m a -> m (Maybe a)
pop (LockStack v) = modifyMVar v (\as -> (drop 1 as, listToMaybe as))

peek :: MonadConc m => LockStack m a -> m (Maybe a)
peek (LockStack v) = fmap listToMaybe (readMVar v)

```

Fig. 3. A lock-based mutable stack.

5 Case Studies

We now present two illustrative case studies: concurrent stacks in Sect. 5.1, and semaphores in Sect. 5.2.

5.1 Concurrent Stacks

Lock-Based Stacks. Mutable stacks are commonly used for synchronisation amongst multiple threads. A simple mutable stack is just an immutable list inside an `MVar` shared variable, as in Fig. 3. We now run CoCo on those functions, where the initialisation function constructs a stack from a list, the observation function converts it back to a list, and the interference function sets the contents of the stack to a given list. CoCo discovers the following properties:

peek @ ->- push x @ >> pop @ (6)

peek @ ->- (push x @) ||| (pop @) (7)

peek @ ->- pop @ >>= \m -> whenJust push @ m (8)

Here `whenJust` is defined as `\f s -> maybe (pure ()) ('f' s)` and `|||` is concurrent composition. Property (6) may seem surprising: the left term returns the top of stack whereas the right term returns the value pushed. Remember that `CoCo` does not consider equality of results when determining properties, only the effect on the state. Property (7) is a consequence of (6). Property (8) is analogous to the `readMVar` properties presented in Sect. 3, as we might expect given how the stack operations are defined.

Buggy Functions. Suppose we add an *incorrect* `push2` function, which is meant to push two values atomically, but which only pushes the second value twice. `CoCo` finds this property:

```
push2 x1 x @ ->- push x @ >> push x @
```

As this is a strict refinement, we now know that `push2` is more deterministic in some way than two `pushes`. As we know that the composition of two `pushes` is not atomic, this strongly suggests that `push2` is. We can also see the effect of `push2` on the state, and that it is incorrect!

```
newtype CASStack m a = CASStack (CRef m [a])

push :: MonadConc m => a -> CASStack m a -> m ()
push a (CASStack r) = modifyCRefCAS r (\as -> (a:as, ()))

pop :: MonadConc m => CASStack m a -> m (Maybe a)
pop (CASStack r) = modifyCRefCAS r (\as -> (drop 1 as, listToMaybe as))

peek :: MonadConc m => CASStack m a -> m (Maybe a)
peek (CASStack r) = fmap listToMaybe (readCRef r)
```

Fig. 4. A lock-free mutable stack.

Choice of Observation. Properties are discovered using a programmer-supplied observation function, so different functions can be used to discover different properties. By changing the observation of our stack from `equality-as-a-list` to `peek`, we discover a new collection of properties. Here we have fixed the `push2` function to behave correctly and also removed `|||` from the signature.

```
peek @ ->- push x @ >> pop @
peek @ === pop @ >>= \m -> whenJust push @ m
push x @ === pop @ >> push x @
push x1 @ === push2 x x1 @ (9)
push x1 @ === push x @ >> push x1 @ (10)
whenJust push @ m === whenJust (push2 x) @ m
```


Properties (9) and (10) show the power of supplying a custom observation function: in the left and right terms, the stack states are *not* equal. In both (9) and (10) the left term increases the stack depth by one, and the right by two. We now see that `push2` leaves its second argument on the top of the stack. We could not directly observe this before, as a single push would leave the stack sizes out of balance. Throwing away unnecessary details, in this case the tail of the stack, allows us to see more than we previously could.

It is important to bear in mind that there is no *best* observation to make, no *best* interference to consider, and no *best* set of properties to discover. Each choice of observation and interference will reveal something about the functions under test. By considering different cases, we can arrive at a fuller understanding of our code.

Choice of Implementation. Due to their blocking behaviour, `MVar`s can have poor performance under contention. An alternative concurrency primitive is the `CRef`,¹ corresponding to a lock-free mutable location in memory. An atomic compare-and-swap operation updates `CRef` values efficiently even with contention. Figure 4 shows our implementation, which is similar to the `MVar` stack.

A feature of `CoCo` that differentiates it from other property-discovery tools is the ability to compare two different signatures which have compatible observation types. We can compare the `MVar` and `CRef` stacks by simply supplying both signatures to the tool, each of which contains `push`, `pop`, `peek`, `whenJust`, and `|||`. `CoCo` then reports 19 properties, including these three:

$$\text{popM } @ \quad === \quad \text{popC } @ \quad (11)$$

$$\text{peekM } @ \quad === \quad \text{peekC } @ \quad (12)$$

$$\text{pushM } x \ @ \quad === \quad \text{pushC } x \ @ \quad (13)$$

Here we use the list observation again. Functions with names ending `M` are for `MVar` stacks, functions with names ending `C` for `CRef` stacks. Properties (11), (12), and (13) tell us what we want to know: the `CRef` stack is equivalent to the `MVar` stack.

A common approach when first writing a program is to do everything in a simple and clearly correct fashion. After checking correctness, we may gradually rewrite components to meet performance requirements. At which point testing must establish that the rewritten components preserve the behaviour. The ability to determine observational equivalence of different implementations of the same API is an alternative to the more-common unit-testing for this task [8].

5.2 Semaphores

A semaphore is a common synchronisation primitive. A semaphore can be thought of as a record of how many units of some abstract resource are available, with operations to adjust the record in a race-free way. *Binary semaphores* only

¹ In regular GHC Haskell this is the `IOWRef`, here *Déjà Fu* deviates from the norm.

have two states, and are used to implement locks. *Counting semaphores* have an arbitrary number of states. An implementation of counting semaphores is provided in the `Control.Concurrent.QSemN` library module.

Figure 5 shows the signature we provide to CoCo. CoCo supports polymorphic function types, as can be seen in the type of `|||`, where `A` and `B` are types we use as type *variables*. The `commLit` function indicates that the supplied binary function is *commutative*, which is used to prune the generated schemas further.

The `new`, `wait`, `signal`, and `remaining` functions are provided by the `QSemN` library module. We construct a new semaphore by allocating an arbitrary amount of resource; we observe how much resource remains; and we interfere by taking and then replacing half of the resource. The interference thread is interleaved with the term thread, so it may cause the term thread to block.

CoCo finds 57 properties in this example, so in the remainder of the section we only discuss selected properties.

Waiting and Signalling. CoCo tells us that the effect of waiting for zero resource and of signalling the availability of zero resource are the same — neither affects the state of the semaphore:

```
wait @ 0 === wait @ 0 >> wait @ 0
signal @ 0 === wait @ 0 >> wait @ 0
```

(14)

Property (14) also shows that waiting for zero resource is not a neutral operation, as if it were CoCo would prune the property away. This suggests that `wait` may block.

```
type C = Concurrency

sig :: Sig (QSemN C) Int Int
sig = Sig
  { initialise = new . abs
  , expressions =
    [ lit "wait" (wait :: QSemN C -> Int -> C ())
    , lit "signal" (signal :: QSemN C -> Int -> C ())
    ]
  , backgroundExpressions =
    [ commLit "|||" ((|||) :: C A -> C B -> C ())
    , commLit "+" ((+) :: Int -> Int -> Int)
    , lit "-" ((-) :: Int -> Int -> Int)
    , lit "0" (0 :: Int)
    , lit "1" (1 :: Int)
    ]
  , interfere = \q n -> let i = n `div` 2 in wait q i >> signal q i
  , observe = \q _ -> remaining q
  , backToSeed = \q _ -> remaining q
  }
```

Fig. 5. CoCo signature for the `QSemN` type.

CoCo also generates properties revealing another implementation detail, that the programmer can `wait` for a negative value instead of calling `signal`:

```
signal @ 1 === wait @ (0 - 1)
signal @ (1 + 1) === wait @ (0 - (1 + 1))
```

We might suspect that the more general property `signal @ x === wait @ (-x)` holds for all positive `x`. CoCo finds this form if we extend our signature with `abs` and `negate`:

```
signal @ (abs x) === wait @ (negate (abs x)) (15)
```

A Lack of Composability. CoCo reports some strict refinements involving `signal` and `wait` where we might expect equivalences:

```
signal @ 0 ->- signal @ x >> wait @ x (16)
```

```
signal @ (x + x1) ->- signal @ x >> signal @ x1 (17)
```

```
signal @ (x + x1) ->- (signal @ x) ||| (signal @ x1) (18)
```

We have just seen with property (15) that funny things happen with negative numbers, so it should be no surprise that these refinements are only equivalences when `x` and `x1` are positive.

Table 2. Scaling behaviour of the semaphore case study.

Term size	1	2	3	4	5	6	7	8
Schemas	15	29	56	88	238	385	1689	2740
Properties	0	0	0	0	1	1	55	55
Time (s)	0.03	0.03	0.45	0.45	9.2	9.2	970	970
Time/Schema ²	1.3e-4	3.6e-5	1.4e-4	5.8e-5	1.6e-4	6.2e-5	3.4e-4	1.3e-4

Types. Signalling or awaiting a negative quantity is a breach of the semaphore protocol. Perhaps a better interface for semaphores would only allow nonnegative quantities. The change might avoid accidental breakage in the future if the semantics of negative values are unwittingly changed.

CoCo supports many types, but not all. If the programmer wishes to use types outside of the built-in collection, they must provide some information: a way to enumerate values, an equality predicate, and a symbol to use in variable names. In this way, the programmer can extend CoCo to work with arbitrary types, or alter the behaviour of existing types. If we have `signal` and `wait` use natural numbers rather than integers, properties (16–18) become equivalences:

```
signal @ 0 === signal @ n >> wait @ n
```

```
signal @ (n + n1) === signal @ n >> signal @ n1
```

```
signal @ (n + n1) === (signal @ n) ||| (signal @ n1)
```

We could pursue this issue further by examining the terms with *Déjà Fu* when given a negative quantity, or we could change the type of the function to forbid that case. Ideally, illegal states should be unrepresentable.

Scaling. Table 2 shows how CoCo scales as the term size increases. The execution time grows rapidly, but the time to compare pairs of schemas So reducing the number of schemas is the most effective way to reduce the execution time. One such area for future improvement is in cases where one schema is an instance of another. Such schemas may arise when the signature includes constants. For example, the schema `signal @ 1` is an instance of `signal @ x`. The ‘most general term’ rule does not apply here, as these are *different* schemas. If CoCo were able to synthesise preconditions, it would be possible in some cases to eliminate constants from signatures, solving this problem.

6 Related Work

QuickSpec and Speculate. The main related work to ours is QuickSpec [3, 12], which automatically discovers equational laws of pure functions by generating schemas and testing terms. The Speculate [1] tool is similar to QuickSpec but in addition can discover *conditional equations* and *inequalities*. Speculate properties may have preconditions, which the tool can find. CoCo does not support conditional equations, but they could be useful. To return to the semaphore case study from Sect. 5.2, the presence of conditional equations would allow us to discover the conditional property $x \geq 0 \implies \text{signal } @ \ x \implies \text{wait } @ \ (\text{negate } x)$, without needing to introduce the `abs` function. Neither QuickSpec nor Speculate support functions with effects or generating lambda-terms.

Bach. The Bach [13] tool uses a database of examples of input/output values from functions to synthesise properties using a Datalog-based oracle. It uses a notion of *evidence* to decide whether an inferred property holds: negative evidence consists of counterexamples; positive evidence consists of witnesses. Bach requires functions to have at most one output for each input, to construct negative evidence. This makes Bach unsuitable for concurrency, which is nondeterministic.

Daikon. The Daikon [6] tool discovers *likely invariants* of C, C++, Java, and Perl programs. It observes variables during program execution, applying machine learning techniques to discover properties. Daikon does not synthesise and test program terms. It is only able to discover invariants which exist in the program. In contrast, the tools mentioned so far, including CoCo, discover properties of combinations of expressions that may not appear in the original program at all.

7 Conclusions and Evaluation

Our broad aim is to help programmers overcome the difficulty of writing correct concurrent programs even in a declarative setting. To work towards this, we have presented a new tool, CoCo, to discover behavioural properties of effectful functions operating on shared state.

Applicability Beyond Haskell. CoCo is tied to Haskell in two ways: it has some knowledge of Haskell types, which is used to generate expressions; and it relies on the Déjà Fu tool to find the results of executing an expression under different schedules. However, it could be reimplemented for another language. For example, in Erlang the objects of interest are processes. Initialisation is to create a process in a known state. Observation is to request information from a process. Interference is to instruct a process to change its internal state. The PULSE tool for systematically testing Erlang programs [2] would play the part of Déjà Fu.

Value of Reported Properties. Although only supported by a finite number of test cases, the properties reported by CoCo are surprisingly accurate in practice. These properties can provide helpful insights into the behaviour of functions. As demonstrated in the semaphore case study (Sect. 5.2), surprising properties can suggest that implementations of some functions rely on unstated assumptions.

Ease of Use. Ideally, a testing tool should not force the programmer to structure their code in a specific way. CoCo requires the use of the concurrency typeclass from Déjà Fu, which is not widespread in practice. However, it has been our experience that reformulating concurrent Haskell code for the necessary abstraction is a type-directed and mechanical process, requiring little insight.

References

1. Braquehais, R., Runciman, C.: Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In: Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Haskell 2017, pp. 40–51. ACM, New York (2017)
2. Claessen, K., Palka, M., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.: Finding race conditions in Erlang with QuickCheck and PULSE. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP 2009, pp. 149–160. ACM (2009)
3. Claessen, K., Smallbone, N., Hughes, J.: QUICKSPEC: guessing formal specifications using testing. In: Fraser, G., Gargantini, A. (eds.) TAP 2010. LNCS, vol. 6143, pp. 6–21. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13977-2_3
4. Coons, K.E., Musuvathi, M., McKinley, K.S.: Bounded partial-order reduction. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2013, pp. 833–848. ACM (2013)
5. 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 (Proc.)* **75**(5), 381–392 (1972)
6. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* **69**(1–3), 35–45 (2007)
7. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, pp. 110–121. ACM (2005)

8. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined resume. In: Robinet, B., Wilhelm, R. (eds.) ESOP 1986. LNCS, vol. 213, pp. 187–196. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-16442-1_14
9. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007, pp. 446–455. ACM (2007)
10. Musuvathi, M., Qadeer, S.: Fair stateless model checking. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008, pp. 362–371. ACM (2008)
11. Jones, S.P., Gordon, A., Finne, S.: Concurrent Haskell. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1996, pp. 295–308. ACM (1996)
12. Smallbone, N., Johansson, M., Claessen, K., Alghed, M.: Quick specifications for the busy programmer. *J. Funct. Program.* **27** (2017)
13. Smith, C., Ferns, G., Albarghouthi, A.: Discovering relational specifications. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, pp. 616–626. ACM (2017)
14. Walker, M., Runciman, C.: Déjà Fu: a concurrency testing library for Haskell. In: Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, pp. 141–152. ACM (2015)

Author Index

- Antoy, Sergio 149
Avanzini, Martin 132
- Cheng, Chen-Mou 68
Cheung, Steven 84
Chitul, Olaf 230
Codish, Michael 182
- Dal Lago, Ugo 132
Darvariu, Victor 84
- Ehlers, Thorsten 182
Emoto, Kento 166
- Frühwirth, Thom 116
- Gall, Daniel 116
Gange, Graeme 182
Ghica, Dan R. 84
- Hamana, Makoto 99
Hanus, Michael 149
Hsu, Ruey-Lin 68
Hu, Zhenjiang 166
- Itzhakov, Avraham 182
Iwasaki, Hideya 166
- Kiselyov, Oleg 33
- Lanese, Ivan 247
- Matsuoka, Satoshi 17
Matsuzaki, Kiminori 166
Mizuno, Masayuki 1
Moriyata, Akimasa 166
Moser, Georg 214
Mu, Shin-Cheng 68
Muroya, Koko 84
- Nishida, Naoki 247
- Palacios, Adrián 247
- Riesco, Adrián 198
Rowe, Reuben N. S. 84
Runciman, Colin 264
- Sakaguchi, Kazuhiko 51
Schneckenreither, Manuel 214
Stuckey, Peter J. 182
Sumii, Eijiro 1
- Tsushima, Kanae 230
- Vidal, Germán 247
- Walker, Michael 264
- Yamada, Akihisa 132