Lecture Notes in Computer Science 2620

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Andrew D. Gordon (Ed.)

Foundations of Software Science and Computation Structures

6th International Conference, FOSSACS 2003
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2003
Warsaw, Poland, April 7-11, 2003
Proceedings

Springer

Andrew D. Gordon
Microsoft Research
7 JJ Thomson Avenue, Cambridge CB3 0FB, UK
E-mail: adg@microsoft.com

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): F.3, F.4.2, F.1.1, D.3.3-4, D.2.1

ISSN 0302-9743
ISBN 3-540-00897-7 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN 10872954 06/3142 5 4 3 2 1 0

Foreword

ETAPS 2003 was the sixth instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised five conferences (FOSSACS, FASE, ESOP, CC, TACAS), 14 satellite workshops (AVIS, CMCS, COCV, FAMAS, Feyerabend, FICS, LDTA, RSKD, SC, TACoS, UniGra, USE, WITS and WOOD), eight invited lectures (not including those that are specific to the satellite events), and several tutorials. We received a record number of submissions to the five conferences this year: over 500, making acceptance rates fall below 30% for every one of them. Congratulations to all the authors who made it to the final program! I hope that all the other authors still found a way of participating in this exciting event and I hope you will continue submitting.

A special event was held to honour the 65th birthday of Prof. Wlad Turski, one of the pioneers of our young science. The deaths of some of our "fathers" in the summer of 2002 — Dahl, Dijkstra and Nygaard — reminded us that Software Science and Technology is, perhaps, no longer that young. Against this sobering background, it is a treat to celebrate one of our most prominent scientists and his lifetime of achievements. It gives me particular personal pleasure that we are able to do this for Wlad during my term as chairman of ETAPS.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a loose confederation in which each event retains its own identity, with a separate program committee and independent proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronized parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for "unifying" talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2003 was organized by Warsaw University, Institute of Informatics, in cooperation with the Foundation for Information Technology Development, as well as:

- European Association for Theoretical Computer Science (EATCS);
- European Association for Programming Languages and Systems (EAPLS);

- European Association of Software Science and Technology (EASST); and
- ACM SIGACT, SIGSOFT and SIGPLAN.

The organizing team comprised:

Mikołaj Bojańczyk, Jacek Chrząszcz, Piotr Chrząstowski-Wachtel, Grzegorz Grudziński, Kazimierz Grygiel, Piotr Hoffman, Janusz Jabłonowski, Mirosław Kowaluk, Marcin Kubica (publicity), Sławomir Leszczyński (www), Wojciech Moczydłowski, Damian Niwiński (satellite events), Aleksy Schubert, Hanna Sokołowska, Piotr Stańczyk, Krzysztof Szafran, Marcin Szczuka, Łukasz Sznuk, Andrzej Tarlecki (co-chair), Jerzy Tiuryn, Jerzy Tyszkiewicz (book exhibition), Paweł Urzyczyn (co-chair), Daria Walukiewicz-Chrząszcz, Artur Zawłocki.

ETAPS 2003 received support from:[1]

- Warsaw University
- European Commission, High-Level Scientific Conferences and Information Society Technologies
- US Navy Office of Naval Research International Field Office,
- European Office of Aerospace Research and Development, US Air Force
- Microsoft Research

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

Egidio Astesiano (Genoa), Pierpaolo Degano (Pisa), Hartmut Ehrig (Berlin), José Fiadeiro (Leicester), Marie-Claude Gaudel (Paris), Evelyn Duesterwald (IBM), Hubert Garavel (Grenoble), Andy Gordon (Microsoft Research, Cambridge), Roberto Gorrieri (Bologna), Susanne Graf (Grenoble), Görel Hedin (Lund), Nigel Horspool (Victoria), Kurt Jensen (Aarhus), Paul Klint (Amsterdam), Tiziana Margaria (Dortmund), Ugo Montanari (Pisa), Mogens Nielsen (Aarhus), Hanne Riis Nielson (Copenhagen), Fernando Orejas (Barcelona), Mauro Pezzè (Milano), Andreas Podelski (Saarbrücken), Don Sannella (Edinburgh), David Schmidt (Kansas), Bernhard Steffen (Dortmund), Andrzej Tarlecki (Warsaw), Igor Walukiewicz (Bordeaux), Herbert Weber (Berlin).

I would like to express my sincere gratitude to all of these people and organizations, the program committee chairs and PC members of the ETAPS conferences, the organizers of the satellite events, the speakers themselves, and Springer-Verlag for agreeing to publish the ETAPS proceedings. The final votes of thanks must go, however, to Andrzej Tarlecki and Paweł Urzyczyn. They accepted the risk of organizing what is the first edition of ETAPS in Eastern Europe, at a time of economic uncertainty, but with great courage and determination. They deserve our greatest applause.

Leicester, January 2003

José Luiz Fiadeiro
ETAPS Steering Committee Chair

[1] The contents of this volume do not necessarily reflect the positions or the policies of these organizations and no official endorsement should be inferred.

Preface

The present volume contains the proceedings of the international conference *Foundations of Software Science and Computation Structures (FOSSACS) 2003*, held in Warsaw, Poland, April 7–9, 2003. FOSSACS is an event of the *Joint European Conferences on Theory and Practice of Software (ETAPS)*. The previous five FOSSACS conferences took place in Lisbon (1998), Amsterdam (1999), Berlin (2000), Genoa (2001), and Grenoble (2002).

FOSSACS presents original papers on foundational research with a clear significance to software science. The Program Committee invited papers on theories and methods to support the analysis, integration, synthesis, transformation, and verification of programs and software systems. We identified the following topics, in particular: algebraic models; automata and language theory; behavioural equivalences; categorical models; computation processes over discrete and continuous data; computation structures; logics of programs; modal, spatial, and temporal logics; models of concurrent, reactive, distributed, and mobile systems; process algebras and calculi; semantics of programming languages; software specification and refinement; transition systems; and type systems and type theory. We received 96 submissions, of which 2 were withdrawn.

This proceedings consists of 27 papers. The first—*A Game Semantics for Generic Polymorphism*—accompanies the invited lecture by Samson Abramsky, University of Oxford. The remaining 26 were selected for publication by the Program Committee during a week-long electronic discussion.

I sincerely thank all the authors of papers submitted to FOSSACS 2003; the number and the quality of papers were exceptionally high this year. Moreover, I would like to thank all the members of the Program Committee for the excellent job they did during a rather demanding selection process, and to thank all our subreferees for their invaluable contributions to this process.

To administer submission and evaluation of papers, we relied on a fine web-based tool provided by METAFrame Technologies, Dortmund; thanks to Martin Karusseit and Tiziana Margaria of METAFrame for their timely support. Finally, thanks are due to the ETAPS 2003 Organizing Committee chaired by Andrzej Tarlecki and Paweł Urzyczyn and to the ETAPS Steering Committee for their efficient coordination of all the activities leading up to FOSSACS 2003.

Cambridge, January 2003 Andrew D. Gordon

Program Committee

Witold Charatonik (Germany and Poland)
Adriana Compagnoni (USA)
Vincent Danos (France)
Andrew D. Gordon (UK, Chair)
Roberto Gorrieri (Italy)
Marta Kwiatkowska (UK)
Eugenio Moggi (Italy)
Uwe Nestmann (Switzerland)
Mogens Nielsen (Denmark)
Flemming Nielson (Denmark)
Francesco Parisi Presicce (Italy)
Dusko Pavlovic (USA)
François Pottier (France)
P.S. Thiagarajan (Singapore)
Igor Walukiewicz (France)
Pierre Wolper (Belgium)

Referees

Samson Abramsky
Luca Aceto
Thorsten Altenkirch
Patrick Baillot
Franco Barbanera
Gilles Barthe
Andrej Bauer
Nick Benton
Martin Berger
Marco Bernardo
Paul Blain-Levy
Eduardo Bonelli
Marcello Bonsangue
Johannes Borgström
Alexis-Julien Bouquet
Julian Bradfield
Mario Bravetti
Sebastien Briais
Antonio Bucciarelli
Mikael Buchholtz
Michele Bugliesi
Nadia Busi
Luis Caires
Luca Cardelli
Franck Cassez
Pietro Cenciarelli
Tom Chothia
Serafino Cicerone
Alessandro Coglio
Ernie Cohen
Flavio Corradini
Vincent Cremet
David Cyrluk
Ferruccio Damiani
Olivier Danvy
Rocco De Nicola
Josee Desharnais
Razvan Diaconescu
Catalin Dima
Theo Dimitrakos
Andreas Dolzmann
Gilles Dowek
Jacques Duparc
Zoltan Esik
Jérôme Feret
Maribel Fernández
Maria Ferreira
Andrzej Filinski
Jean-Christophe Filliâtre
Marcelo Fiore
Fabien Fleutot
Riccardo Focardi
Martin Fränzle
Roberto Giacobazzi
Jean Goubault-Larrecq
Erich Grädel
Maria Grazia Vigliotti
Martin Grohe
Dimitar Guelev
Michael Hansen
Anne Haxthausen
Matthew Hennessy

Martin Henz
Hugo Herbelin
Ralph Hinze
Martin Hofmann
Furio Honsell
Hans Hüttel
Dominic Hughes
Michael Huth
Radha Jagadeesan
Rosa Jimenez
Achim Jung
Marcin Jurdzinski
Fairouz Kamareddine
Delia Kesner
Ekkart Kindler
Bartek Klin
Maciej Koutny
Ralf Kuesters
Dietrich Kuske
Anna Labella
Cosimo Laneve
Olivier Laurent
Julia Lawall
James Leifer
Xavier Leroy
Roberto Lucchi
Gerald Lüttgen
P. Madhusudan
Jerzy Marcinkowski
Simon Marlow
Fabio Martinelli
Conor McBride
Cathy Meadows
Lambert Meertens
Paul-André Melliès
Massimo Merro
Marino Miculan
Dale Miller
Marcin Młotkowski
Anders Møller
Peter D. Mosses
Andrzej Murawski
Anca Muscholl
Raja Nagarajan
Monica Nesi
Lasse Nielsen
Gethin Norman
Peter Padawitz
Catuscia Palamidessi
Prakash Panangaden
Antonio Piccolboni
Elvira Pino
Adolfo Piperno
Marc Pouzet
John Power
R. Ramanujam
Uday Reddy
Didier Rémy
Arend Rensink
Marina Ribaudo
Eike Ritter
Christine Röckl
Abhik Roychoudhury
Rene Rydhof Hansen
Andrei Sabelfeld
Amr Sabry
Ivano Salvo
Michel Schinz
Alan Schmitt
Gunnar Schröter
Aleksy Schubert
Peter Sewell
Carron Shankland
Marta Simeoni
Vincent Simonet
Sebastian Skalberg
Kostas Skandalis
Doug Smith
Pawel Sobocinski
Jin Song Dong
Zdzislaw Splawski
Jiří Srba
Rick Statman
Asuman Suenbuel
Martin Sulzmann
Wing-Kin Sung
Maciej Szreter
Jean-Marc Talbot
Andrzej Tarlecki
Lidia Tendera

Hayo Thielecke
Wolfgang Thomas
Simone Tini
Tayssir Touili
Tomasz Truderung
Mark Utting
Frank Valencia
Femke van Raamsdonk
Daniele Varacca
Joe Wells
Stephen Westfold
Pawel Wojciechowski
James Worrell
Gianluigi Zavattaro
Wiesław Zielonka
Pascal Zimmer
Lenore Zuck

Table of Contents

Invited Paper

Contributed Papers

A Game Semantics for Generic Polymorphism

Samson Abramsky[1⋆] and Radha Jagadeesan[2⋆⋆]

[1] Oxford University Computing Laboratory
samson@comlab.ox.ac.uk
[2] DePaul University
rjagadeesan@cs.depaul.edu

Abstract. Genericity is the idea that the same program can work at many different data types. Longo, Milsted and Soloviev proposed to capture the inability of generic programs to probe the structure of their instances by the following equational principle: if two generic programs, viewed as terms of type $\forall X.\, A[X]$, are equal at any given instance $A[T]$, then they are equal at all instances. They proved that this rule is admissible in a certain extension of System F, but finding a semantically motivated model satisfying this principle remained an open problem.
In the present paper, we construct a categorical model of polymorphism, based on game semantics, which contains a large collection of generic types. This model builds on two novel constructions:

- A direct interpretation of variable types as games, with a natural notion of substitution of games. This allows moves in games $A[T]$ to be decomposed into the generic part from A, and the part pertaining to the instance T. This leads to a simple and natural notion of generic strategy.
- A "relative polymorphic product" $\Pi_i(A, B)$ which expresses quantification over the type variable X_i in the variable type A with respect to a "universe" which is explicitly given as an additional parameter B. We then solve a recursive equation involving this relative product to obtain a universe in a suitably "absolute" sense.

Full Completeness for ML types (universal closures of quantifier-free types) can be proved for this model.

1 Introduction

We begin with an illuminating quotation from Gérard Berry [9]:

> Although it is not always made explicit, the *Write Things Once* or WTO principle is clearly the basis for loops, procedures, higher-order functions, object-oriented programming and inheritance, concurrency *vs.* choice between interleavings, etc.

⋆ Samson Abramsky was supported in part by UK EPSRC.
⋆⋆ Radha Jagadeesan was supported in part by NSF CCR-020244901.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 1–22, 2003.

In short, much of the search for high-level structure in programming can be seen as the search for concepts which allow commonality to be expressed. An important facet of this quest concerns *genericity*: the idea that the same program can work at many different data types.

For illustration, consider the abstraction step involved in passing from list-processing programs which work on data types $\mathsf{List}[T]$ for specific types T, to programs which work generically on $\mathsf{List}[X]$. Since lists can be so clearly visualized, it is easy to see what this should mean (see Figure 1). A generic program cannot probe the internal structure of the list elements. Thus e.g. list concatenation and reversal are generic, while summing a list is not. However, when we go beyond lists and other concrete data structures, to higher-order types and beyond, what genericity or type-independence should mean becomes much less clear.

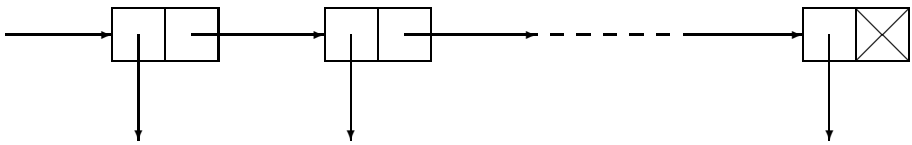


Fig. 1. 'Generic' list structure

One very influential proposal for a general understanding of the *uniformity* which generic programs should exhibit with respect to the type instances has been John Reynolds' notion of *relational parametricity* [21], which requires that relations between instances be preserved in a suitable sense by generic programs. This has led to numerous further developments, e.g. [17,1,19].

Relational parametricity is a beautiful and important notion. However, in our view it is not the whole story. In particular:

- It is a "pointwise" notion, which gets at genericity indirectly, via a notion of uniformity applied to the family of instantiations of the program, rather than directly capturing the idea of a program written at the generic level, which necessarily cannot probe the structure of an instance.
- It is closely linked to strong extensionality principles, as shown e.g. in [1,19], whereas the intuition of generic programs not probing the structure of instances is *prima facie* an intensional notion—a constraint on the behaviour of processes.

An interestingly different analysis of genericity with different formal consequences was proposed by Giuseppe Longo, Kathleen Milsted and Sergei Soloviev [15,16]. Their idea was to capture the inability of generic programs to probe the structure of their instances by the following equational principle: if two generic programs,

viewed as terms t, u of type $A[X]$, are equal at *any* given instance T, then they are equal at *all* instances:

$$\exists T.\, t\{T\} = u\{T\} : A[T] \implies \forall U.\, t\{U\} = u\{U\} : A[U].$$

This principle can be stated even more strongly when second-order polymorphic quantification over type variables is used. For $t, u : \forall X.\, A$:

$$\frac{t\{T\} = u\{T\} : A[T]}{t = u : \forall X.\, A}.$$

We call this the *Genericity Rule*. In one of the most striking syntactic results obtained for System F (*i.e.* the polymorphic second-order λ-calculus [10,20]), Longo, Milsted and Soloviev proved in [15] that the Genericity Rule is admissible in the system obtained by extending System F with the following axiom scheme:

$$(C) \qquad t\{B\} = t\{C\} : A \qquad (t : \forall X.\, A,\ X \notin \mathrm{FV}(A)).$$

While many of the known semantic models of System F satisfy axiom (C), *there is no known naturally occurring model which satisfies the Genericity principle* (*i.e.* in which the rule of Genericity is valid). In fact, in the strong form given above, the Genericity rule is actually *incompatible* with well-pointedness and parametricity, as observed by Longo. Thus if we take the standard polymorphic terms representing the Boolean values

$$\Lambda X.\, \lambda x{:}X.\, \lambda y{:}X.\, x, \quad \Lambda X.\, \lambda x{:}X.\, \lambda y{:}X.\, y \quad : \quad \forall X.\, X \to X \to X$$

then if the type $\forall X.\, X \to X$ has only one inhabitant — as will be the case in a parametric model — then by well-pointedness the Boolean values will be equated at this instance, while they cannot be equated in general on pain of inconsistency.

However, we can state a more refined version. Say that a type T is *a generic instance* if for all types $A[X]$:

$$t\{T\} = u\{T\} : A[T] \implies t = u : \forall X.\, A.$$

This leads to the following problem posed by Longo in [16], and still, to the best of our knowledge, open:

> Open Problem 2. Construct, at least, some (categorical) models that contain a collection of "generic" types. ... If our intuition about constructivity is correct, infinite objects in categories of (effective) sets should satisfy this property.

In the present paper, we present a solution to this problem by constructing a categorical model of polymorphism which contains a large collection of generic types. The model is based on game semantics; more precisely, it extends the "AJM games" of [5] to provide a model for generic polymorphism. Moreover, Longo's intuition as expressed above is confirmed in the following sense: our main

sufficient condition for games (as denotations of types) to be generic instances is that they have plays of arbitrary length. This can be seen as an intensional version of Longo's intuition about infinite objects.

In addition to providing a solution to this problem, the present paper also makes the following contributions.

- We interpret variable types in a simple and direct way, with a natural notion of *substitution of games into variable games*. The crucial aspect of this idea is that it allows moves in games $A[T]$ to be decomposed into the generic part from A, and the part pertaining to the instance T. This in turn allows the evident content of genericity in the case of concrete data structures such as lists to be carried over to arbitrary higher-order and polymorphic types. In particular, we obtain a simple and natural notion of *generic strategy*. This extends the notion of history-free strategy from [5], which is determined by a function on moves, to that of a generic strategy, which is determined by a function on *the generic part of the move only*, and simply acts as the identity on the part pertaining to the instance. This captures the intuitive idea of a generic program, existing "in advance" of its instances, in a rather direct way.
- We solve the size problem inherent in modelling System F in a somewhat novel way. We define a "relative polymorphic product" $\Pi_i(A, B)$ which expresses quantification over the type variable X_i in the variable type A with respect to a "universe" which is explicitly given as an additional parameter B. We then solve a recursive equation involving this relative product to obtain a universe in a suitably "absolute" sense: a game $\mathcal{U}$ with the requisite closure properties to provide a model for System F.

It is also possible to prove a Full Completeness theorem for the ML types (*i.e.* the universal closures of quantifier-free types). For this, further technical details, and proofs of the results, we refer to the full version of this paper [4].

2 Background

2.1 Syntax of System F

We briefly review the syntax of System F. For further background information we refer to [11].

Types (Formulas)

$$A \quad ;;= \quad X \mid A \to B \mid \forall X.\, A$$

Typing Judgements Terms in context have the form

$$x_1 : A_1, \ldots, x_k : A_k \vdash t : A$$

Assumption

$$\frac{}{\Gamma, x : t \vdash x : T}$$

Implication

$$\frac{\Gamma, x : U \vdash t : T}{\Gamma \vdash \lambda x{:}U.\, t : U \to T}\ (\to - I) \qquad \frac{\Gamma \vdash t : U \to T \qquad \Gamma \vdash u : U}{\Gamma \vdash tu : T}\ (\to - E)$$

Second-order Quantification

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \Lambda X.\, t : \forall X.\, A}\ (\forall - I) \qquad \frac{\Gamma \vdash t : \forall X.\, A}{\Gamma \vdash t\{B\} : A[B/X]}\ (\forall - E)$$

The $(\forall - I)$ rule is subject to the usual eigenvariable condition, that X does not occur free in Γ.
The following isomorphism is definable in System F:

$$\forall X.\, A \to B \ \ \cong \ \ A \to \forall X.\, B \qquad (X \notin \mathrm{FV}(A)).$$

This allows us to use the following normal form for types:

$$\forall \boldsymbol{X}.\, T_1 \to \cdots \to T_k \to X \qquad (k \geq 0)$$

where each T_i is inductively of the same form.

2.2 Notation

We write ω for the set of natural numbers.

We shall use vector notation, writing $\boldsymbol{A}$ for a list $A_1, \ldots, A_k$.

If X is a set, X^* is the set of finite sequences (words, strings) over X. We use s, t, u, v to denote sequences, and a, b, c, d, m, n to denote elements of these sequences. Concatenation of sequences is indicated by juxtaposition, and we don't distinguish notationally between an element and the corresponding unit sequence. Thus as denotes the sequence with first element a and tail s. However, we will sometimes write $a \cdot s$ or $s \cdot a$ to give the name a to the first or last element of a sequence.

If $f : X \longrightarrow Y$ then $f^* : X^* \longrightarrow Y^*$ is the unique monoid homomorphism extending f. We write $|s|$ for the length of a finite sequence, and s_i for the ith element of s, $1 \leq i \leq |s|$. We write $\mathsf{numoccs}(a, s)$ for the number of occurrences of a in the sequence s.

We write $s \sqsubseteq t$ if s is a prefix of t, *i.e.* $t = su$ for some u. We write $s \sqsubseteq^{\mathrm{even}} t$ if s is an even-length prefix of t. $\mathsf{Pref}(S)$ is the set of prefixes of elements of $S \subseteq X^*$. S is *prefix-closed* if $S = \mathsf{Pref}(S)$.

3 Variable Games and Substitution

3.1 A Universe of Moves

We fix an algebraic signature consisting of the following set of *unary* operations:

$$\mathtt{p},\ \mathtt{q},\ \{\mathtt{l}_i \mid i \in \omega\},\ \mathtt{r}.$$

We take $\mathcal{M}$ to be the algebra over this signature freely generated by ω. Explicitly, $\mathcal{M}$ has the following "concrete syntax":

$$m \quad ::= \quad i\ (i \in \omega) \quad | \quad \mathtt{p}(m) \quad | \quad \mathtt{q}(m) \quad | \quad \mathtt{l}_i(m)\ \ (i \in \omega) \quad | \quad \mathtt{r}(m).$$

For any algebra $(A, \mathtt{p}^A, \mathtt{q}^A, \{\mathtt{l}_i^A \mid i \in \omega\}, \mathtt{r}^A)$ and map $f : \omega \longrightarrow A$, there is a unique homomorphism $f^\dagger : \mathcal{M} \longrightarrow A$ extending f, defined by:

$$f^\dagger(i) = f(i), \qquad f^\dagger(\phi(m)) = \phi^A(f^\dagger(m)) \quad (\phi \in \{\mathtt{p}, \mathtt{q}, \mathtt{r}\} \cup \{\mathtt{l}_i \mid i \in \omega\}).$$

We now define a number of maps on $\mathcal{M}$ by this means.

- The *labelling map* $\lambda : \mathcal{M} \longrightarrow \{P, O\}$. The polarity algebra on the carrier $\{P, O\}$ interprets $\mathtt{p}$, $\mathtt{q}$, $\mathtt{r}$ as the identity, and each $\mathtt{l}_i$ as the involution $\bar{(\ )}$, where $\bar{P} = O$, $\bar{O} = P$. The map on the generators is the constant map sending each i to O.
- The map $\rho : \mathcal{M} \longrightarrow \omega$ sends each move to the unique generator occurring in it. All the unary operations are interpreted as the identity, and the map on generators is the identity.
- The substitution map. For each move $m' \in \mathcal{M}$, there is a map

  $$h_{m'} : \mathcal{M} \longrightarrow \mathcal{M}$$

  induced by the constant map on ω which sends each i to m'. We write $m[m']$ for $h_{m'}(m)$.
- An alternative form of substitution is written $m[m'/i]$. This is induced by the map which send i to m', and is the identity on all $j \neq i$.

Proposition 1. *Substitution interacts with λ and ρ as follows.*

1. $\lambda(m[m']) = \begin{cases} \overline{\lambda(m')} \text{ if } \lambda(m) = P \\ \lambda(m') \text{ if } \lambda(m) = O \end{cases}$
2. $\rho(m[m']) = \rho(m')$.

We extend the notions of substitution pointwise to sequences and sets of sequences of moves in the evident fashion.
We say that $m_1, m_2 \in \mathcal{M}$ are *unifiable* if for some $m_3, m_4 \in \mathcal{M}$, $m_1[m_3] = m_2[m_4]$. A set $S \subseteq \mathcal{M}$ is *unambiguous* if whenever $m_1, m_2 \in S$ are unifiable, $m_1 = m_2$.

Given a subset $S \subseteq \mathcal{M}$ and $i \in \omega$, we write

$$S^i = \{m \in S \mid \rho(m) = i\}.$$

We define a notion of projection of a sequence of moves s onto a move m inductively as follows:

$$\begin{array}{lll} \varepsilon \upharpoonright m & = \varepsilon & \\ m[m'] \cdot s \upharpoonright m & = m' \cdot (s \upharpoonright m) & \\ m' \cdot s \upharpoonright m & = s \upharpoonright m, & \forall m''.\, m' \neq m[m'']. \end{array}$$

Dually, given an unambiguous set of moves S, and a sequence of moves s in which every move has the form $m[m']$ for some $m \in S$ (necessarily unique since S is unambiguous), we define a projection $s \upharpoonright S$ inductively as follows:

$$\begin{array}{lll} \varepsilon \upharpoonright S & = \varepsilon & \\ m[m'] \cdot s \upharpoonright S & = m \cdot (s \upharpoonright S) & (m \in S \ \wedge \ \rho(m) > 0) \\ m[m'] \cdot s \upharpoonright S & = m[m'] \cdot (s \upharpoonright S) & (m \in S^0) \end{array}$$

3.2 Variable Games

A *variable game* is a structure

$$A = (\mathcal{O}_A, P_A, \approx_A)$$

where:

- $\mathcal{O}_A \subseteq \mathcal{M}$ is an unambiguous set of moves: the *occurrences* of A. We then define:
  - $\lambda_A = \lambda \upharpoonright \mathcal{O}_A$.
  - $\rho_A = \rho \upharpoonright \mathcal{O}_A$.
  - $M_A = \{m[m'] \mid m \in \mathcal{O}_A^0 \ \wedge \ m' \in \mathcal{M}\} \ \cup \ \bigcup_{j>0} \mathcal{O}_A^j$.
- P_A is a non-empty prefix-closed subset of M_A^* satisfying the following form of *alternation condition*: the odd-numbered moves in a play are *moves by* O, while the even-numbered moves are *by* P. Here we regard the first, third, fifth, ... occurrences of a move m in a sequence as being by $\lambda_A(m)$, while the second, fourth, sixth ... occurrences are by the other player.
- $\approx_A$ is an equivalence relation on P_A such that:

$$\begin{array}{ll} (\mathbf{e1}) & s \approx_A t \implies s \longleftrightarrow t \\ (\mathbf{e2}) & ss' \approx_A tt' \ \wedge \ |s| = |t| \implies s \approx_A t \\ (\mathbf{e3}) & s \approx_A t \ \wedge \ sa \in P_A \implies \exists b.\, sa \approx_A tb. \end{array}$$

  Here $s \longleftrightarrow t$ holds if

$$s = \langle m_1, \ldots, m_k \rangle, \qquad t = \langle m'_1, \ldots, m'_k \rangle$$

  and the correspondence $m_i \longleftrightarrow m'_i$ is bijective and preserves λ_A and ρ_A. We write

$$\pi : s \longleftrightarrow t$$

  to give the name π to the bijective correspondence $m_i \longleftrightarrow m'_i$.

A move $m \in \mathcal{O}_A^i$, $i > 0$, is an *occurrence* of the type variable X_i, while $m \in \mathcal{O}_A^0$ is a *bound occurrence*.

The set of variable games is denoted by $\mathcal{G}(\omega)$. The set of those games A for which the range of ρ_A is included in $\{0, \ldots, k\}$ is denoted by $\mathcal{G}(k)$. Note that if $k \leq l$, then

$$\mathcal{G}(k) \subseteq \mathcal{G}(l) \subseteq \mathcal{G}(\omega).$$

$\mathcal{G}(0)$ is the set of *closed games*.

Comparison with AJM games The above definition of game differs from that in [5] in several minor respects.

1. The notion of bracketing condition, requiring a classification of moves as *questions* or *answers*, has been omitted. This is because we are dealing here with pure type theories, with no notion of "ground data types".
2. The alternation condition has been modified: we still have strict OP-alternation of moves, but now successive occurrences of moves within a sequence are regarded as themselves having alternating polarities. Since in the PCF games in [5] moves in fact only occur once in any play, they do fall within the present formulation. The reason for the revised formulation is that moves in variable games are to be seen as *occurrences* of type variables, which can be expanded into plays at an instance.
3. We have replaced the condition (**e1**) from [5] with a slightly stronger condition, which is in fact satisfied by the games in [5].

3.3 Constructions on Games

Since variable games are essentially just AJM games with some additional structure on moves, the cartesian closed structure on AJM games can be lifted straighforwardly to variable games.

Unit Type The unit type $\mathbf{1}$ is the empty game.

$$\mathbf{1} \;=\; (\varnothing, \{\varepsilon\}, \{(\varepsilon, \varepsilon)\}).$$

Product The product $A\&B$ is the disjoint union of games.

$$\mathcal{O}_{A\&B} = \{\mathtt{p}(m) \mid m \in \mathcal{O}_A\} \cup \{\mathtt{q}(m) \mid m \in \mathcal{O}_B\}$$

$$P_{A\&B} = \{\mathtt{p}^*(s) \mid s \in P_A\} \cup \{\mathtt{q}^*(t) \mid t \in P_B\}$$

$$\mathtt{p}^*(s) \approx_{A\&B} \mathtt{p}^*(t) \;\equiv\; s \approx_A t \qquad \mathtt{q}^*(s) \approx_{A\&B} \mathtt{q}^*(t) \;\equiv\; s \approx_B t.$$

Function Space The function space $A \Rightarrow B$ is defined as follows.

$$\mathcal{O}_{A\Rightarrow B} \;=\; \{\mathtt{l}_i(m) \mid i \in \omega \;\wedge\; m \in \mathcal{O}_A\} \;\cup\; \{\mathtt{r}(m) \mid m \in \mathcal{O}_B\}.$$

$P_{A\Rightarrow B}$ is defined to be the set of all sequences in $M^*_{A\Rightarrow B}$ satisfying the alternation condition, and such that:

- $\forall i \in \omega.\, s{\restriction}\mathtt{l}_i(1) \in P_A$.
- $s{\restriction}\mathtt{r}(1) \in P_B$.

Let $S = \{\mathtt{l}_i(1) \mid i \in \omega\} \cup \{\mathtt{r}(1)\}$. Note that S is unambiguous. Given a permutation α on ω, we define

$$\breve{\alpha}(\mathtt{l}_i(1)) = \mathtt{l}_{\alpha(i)}(1), \qquad \breve{\alpha}(\mathtt{r}(1)) = \mathtt{r}(1).$$

The equivalence relation $s \approx_{A\Rightarrow B} t$ is defined by the condition

$$\exists \alpha \in S(\omega).\, \breve{\alpha}^*(s{\restriction}S) = t{\restriction}S \;\wedge\; s{\restriction}\mathtt{r}(1) \approx_B t{\restriction}\mathtt{r}(1) \;\wedge\; \forall i \in \omega.\, s{\restriction}\mathtt{l}_i(1) \approx_A t{\restriction}\mathtt{l}_{\alpha(i)}(1)).$$

This is essentially identical to the definition in [5]. The only difference is that we use the revised version of the alternation condition in defining the positions, and that we define $A \Rightarrow B$ directly, rather than via the linear connectives $\multimap$ and !.

3.4 Substitution

Given $A \in \mathcal{G}(k)$, and $\boldsymbol{B} = B_1, \ldots, B_k \in \mathcal{G}(l)$, we define $A[\boldsymbol{B}] \in \mathcal{G}(l)$ as follows.

$$\mathcal{O}_{A[\boldsymbol{B}]} \;=\; \mathcal{O}^0_A \;\cup\; \bigcup_{i=1}^{k} \{m[m'] \mid m \in \mathcal{O}^i_A \;\wedge\; m' \in \mathcal{O}_{A_i}\}.$$

$$P_{A[\boldsymbol{B}]} = \{s \in M^*_{A[\boldsymbol{B}]} \mid s{\restriction}A \in P_A \;\wedge\; \forall i : 1 \le i \le k.\, \forall m \in \mathcal{O}^i_A.\, s{\restriction}m \in P_{B_i}\}$$

$$\begin{aligned} s \approx_{A[\boldsymbol{B}]} t \;\equiv\; & s{\restriction}A \approx_A t{\restriction}A \\ & \wedge \\ & \pi : s{\restriction}A \longleftrightarrow t{\restriction}A \implies \forall i : 1 \le i \le k.\, \forall m \in \mathcal{O}^i_A.\, s{\restriction}m \approx_{B_i} t{\restriction}\pi(m). \end{aligned}$$

Here by convenient abuse of notation we write $s{\restriction}A$ for $s{\restriction}\mathcal{O}_A$.

Note that the above definitions would still make sense if we took $k = \omega$ and/or $l = \omega$, so that, for example, there is a well-defined operation

$$\mathcal{G}(\omega) \times \mathcal{G}(\omega)^\omega \longrightarrow \mathcal{G}(\omega).$$

In practice, the finitary versions will be more useful for our purposes here, as they correspond to the finitary syntax of System F.

3.5 Properties of Substitution

Proposition 2. *If $A \in \mathcal{G}(k)$, $B_1, \ldots, B_k \in \mathcal{G}(l)$, and $C_1, \ldots, C_l \in \mathcal{G}(m)$, then:*

$$A[B_1[\boldsymbol{C}], \ldots, B_k[\boldsymbol{C}]] \;=\; (A[B_1, \ldots, B_k])[\boldsymbol{C}].$$

For each $i > 0$ we define the variable game X_i as follows.

$$\begin{array}{ll} \mathcal{O}_{X_i} & = \{i\} \\ P_{X_i} & = M^*_{X_i} \\ s \approx_{X_i} t & \equiv s = t \end{array}$$

Proposition 3. *1. For all $B_1, \ldots B_k \in \mathcal{G}(\omega)$, $i \leq k$: $X_i[B_1, \ldots B_k] \;=\; B_i$.*
2. For all $A \in \mathcal{G}(k)$: $A[X_1, \ldots, X_k] \;=\; A$.

We can define a useful variant of substitution by:

$$A[B/X_i] \;\;=\;\; A[X_1, \ldots, X_{i-1}, B, X_{i+1}, \ldots, X_k]$$

for $A \in \mathcal{G}(k)$, $1 \leq i \leq k$.

Proposition 4. *The cartesian closed structure commutes with substitution:*

$$\begin{array}{l} 1.\ (A \Rightarrow B)[\boldsymbol{C}] = A[\boldsymbol{C}] \Rightarrow B[\boldsymbol{C}] \\ 2.\ (A \,\&\, B)[\boldsymbol{C}] \;=\; A[\boldsymbol{C}] \,\&\, B[\boldsymbol{C}]. \end{array}$$

Combining Propositions 3 and 4, we obtain:

Proposition 5. *The cartesian closed constructions can be obtained by substitution from their generic forms:*

$$\begin{array}{l} 1.\ A \Rightarrow B = X_1 \Rightarrow X_2[A, B] \\ 2.\ A \,\&\, B \;=\; X_1 \,\&\, X_2[A, B]. \end{array}$$

4 Constructing a Universe for Polymorphism

4.1 The Inclusion Order

We define $A \trianglelefteq B$ by:

- $\mathcal{O}_A \subseteq \mathcal{O}_B$
- $P_A \subseteq P_B$
- $s \approx_A t \iff s \in P_A \;\wedge\; s \approx_B t$

The inclusion order is useful in the following context. Suppose we fix a "big game" $\mathcal{U}$ to serve as a "universe". Define a *sub-game* of $\mathcal{U}$ to be a game of the form

$$A = (\mathcal{O}_{\mathcal{U}}, P_A, \approx_{\mathcal{U}} \cap P_A^2),$$

where $P_A \subseteq P_{\mathcal{U}}$, and

$$s \in P_A \ \wedge \ s \approx_{\mathcal{U}} t \ \implies \ t \in P_A.$$

Thus sub-games of $\mathcal{U}$ are completely determined by their sets of positions. We write $\mathsf{Sub}(\mathcal{U})$ for the set of sub-games of $\mathcal{U}$. Note that, for $A, B \in \mathsf{Sub}(\mathcal{U})$:

$$A \trianglelefteq B \iff P_A \subseteq P_B.$$

Proposition 6. *1. $\mathsf{Sub}(\mathcal{U})$ is a complete lattice, with meets and joins given by intersections and unions respectively.*
2. If $S \subseteq P_{\mathcal{U}}$, then the least sub-game $A \in \mathsf{Sub}(\mathcal{U})$ such that $S \subseteq P_A$ is defined by

$$P_A = \{u \mid \exists s \in S.\, \exists t.\, t \sqsubseteq s \ \wedge \ u \approx_{\mathcal{U}} t\}.$$

It is straightforward to verify that function space and product are monotonic with respect to the inclusion order. This leads to the following point, which will be important for our model construction.

Proposition 7. *Suppose that $\mathcal{U}$ is such that*

$$\mathcal{U} \Rightarrow \mathcal{U} \trianglelefteq \mathcal{U}, \qquad \mathcal{U} \,\&\, \mathcal{U} \trianglelefteq \mathcal{U}, \qquad \mathbf{1} \trianglelefteq \mathcal{U}.$$

Then $\mathsf{Sub}(\mathcal{U})$ is closed under these constructions.

Adjoints of substitution Let A be a variable game, and $s \in P_{A[\mathcal{U}/X_i]}$. We can use the substitution structure to compute the *least* instance B (with respect to $\trianglelefteq$) such that $s \in P_{A[B/X_i]}$. We define

$$A_i^*(s) = \{t \mid \exists u.\, \exists m \in \mathcal{O}_A^i.\, t \approx u \ \wedge \ u \sqsubseteq s\restriction m\}$$

Proposition 8. *With notation as in the preceding paragraph, let $B = A_i^*(s)$.*

1. $s \in P_{A[B/X_i]}$.
2. $s \in P_{A[C/X_i]} \implies B \trianglelefteq C$.

4.2 The Relative Polymorphic Product

Given $A, B \in \mathcal{G}(\omega)$ and $i > 0$, we define the relative polymorphic product $\Pi_i(A, B)$ (the "second-order quantification over X_i in the variable type A relative to the universe B") as follows.

$$\mathcal{O}_{\Pi_i(A,B)} = \mathcal{O}_A[0/i] = \{m[0/i] \mid m \in \mathcal{O}_A\}.$$

$$P_{\Pi_i(A,B)} = \{s \in P_{A[B/X_i]} \mid \forall t \cdot a \sqsubseteq^{\text{even}} s.\ A_i^*(t \cdot a) = A_i^*(t)\}$$

$$s \approx_{\Pi_i(A,B)} t \iff s \approx_{A[B/X_i]} t.$$

To understand the definition of $P_{\Pi_i(A,B)}$, it is helpful to consider the following alternative, inductive definition (*cf.* [2]):

$$\begin{aligned} P_{\Pi_i(A,B)} = \quad & \{\epsilon\} \\ & \cup \{sa \mid s \in P^{\text{even}}_{\Pi_i(A,B)} \ \wedge \ \exists C \in \mathsf{Sub}(B).\, sa \in P_{A[C]}\} \\ & \cup \{sab \mid sa \in P^{\text{odd}}_{\Pi_i(A,B)} \ \wedge \ \forall C \in \mathsf{Sub}(B).\, sa \in P_{A[C]} \Rightarrow sab \in P_{A[C]}\} \end{aligned}$$

The first clause in the definition of $P_{\Pi(F)}$ is the basis of the induction. The second clause refers to positions in which it is Opponent's turn to move. It says that Opponent may play in any way which is valid in *some* instance. The final clause refers to positions in which it is Player's turn to move. It says that Player can only move in a fashion which is valid in *every* possible instance. The equivalence of this definition to the one given above follows easily from Proposition 8.

Intuitively, this definition says that initially, nothing is known about which instance we are playing in. Opponent progressively reveals the "game board" ; at each stage, Player is constrained to play within the instance *thus far revealed* by Opponent.

The advantage of the definition we have given above is that it avoids quantification over subgames of B in favour of purely local conditions on the plays.

Proposition 9. *The relative polymorphic product commutes with substitution.*

1. $\Pi_i(A,B)[C/X_i] = \Pi_i(A,B)$.
2. *If* $A \in \mathcal{G}(k+1)$ *and* $C_1, \ldots, C_k \in \mathcal{G}(n)$, *then:*

$$\Pi_{k+1}(A,B)[\boldsymbol{C}] \ = \ \Pi_{n+1}(A[\boldsymbol{C}, X_{n+1}], B).$$

4.3 A Domain Equation for System F

We define a variable game $\mathcal{U} \in \mathcal{G}(\omega)$ of System F types by the following recursive equation:

$$\mathcal{U} \quad = \quad \&_{i>0} X_i \ \& \ \mathbf{1} \ \& \ (\mathcal{U} \,\&\, \mathcal{U}) \ \& \ (\mathcal{U} \Rightarrow \mathcal{U}) \ \& \ \&_{i>0} \Pi_i(\mathcal{U}, \mathcal{U}).$$

Explicitly, $\mathcal{U}$ is being defined as the least fixed point of a function $F : \mathcal{G}(\omega) \longrightarrow \mathcal{G}(\omega)$. The theory developed in [8] can be used to guarantee the existence of this least fixpoint.

We can then define second-order quantification by:

$$\forall X_i.\, A \ \triangleq \ \Pi_i(A, \mathcal{U}).$$

Although it is not literally the case that

$$X_i \trianglelefteq \mathcal{U}, \qquad \mathcal{U} \Rightarrow \mathcal{U} \trianglelefteq \mathcal{U}, \qquad \text{etc.}$$

for trivial reasons of how disjoint union is defined, with a little adjustment of definitions we can arrange things so that we indeed have

$$\begin{aligned} &\bullet && X_i \trianglelefteq \mathcal{U} \\ &\bullet && \mathbf{1} \trianglelefteq \mathcal{U} \\ &\bullet\ A, B \trianglelefteq \mathcal{U} \Longrightarrow && A \,\&\, B \trianglelefteq \mathcal{U} \,\&\, \mathcal{U} \trianglelefteq \mathcal{U} \\ &\bullet\ A, B \trianglelefteq \mathcal{U} \Longrightarrow && A \Rightarrow B \trianglelefteq \mathcal{U} \Rightarrow \mathcal{U} \trianglelefteq \mathcal{U} \\ &\bullet\ A \trianglelefteq \mathcal{U} \quad \Longrightarrow && \forall X_i.\, A = \Pi_i(A, \mathcal{U}) \trianglelefteq \Pi_i(\mathcal{U}, \mathcal{U}) \trianglelefteq \mathcal{U}. \end{aligned}$$

Thus we get a direct inductive definition of the types of System F as sub-games of $\mathcal{U}$.

Moreover, if A and B are (the variable games corresponding to) System F types, then a simple induction on the structure of A using Propositions 3, 4 and 9 shows that

$$A[B/X_i] \trianglelefteq \mathcal{U},$$

and similarly for simultaneous substitution.

5 Strategies

Fix a variable game A. Let

$$g : \mathcal{O}_A \longrightarrow\!\!\!\!\!\cdot\ \ \mathcal{O}_A$$

be a partial function. We can extend g to a partial function

$$\hat{g} : M_{A[\boldsymbol{\mathcal{U}}]} \longrightarrow\!\!\!\!\!\cdot\ \ M_{A[\boldsymbol{\mathcal{U}}]}$$

by

$$\hat{g}(m[m']) \ = \ \begin{cases} g(m)[m'], \ g(m) \text{ defined} \\ \text{undefined otherwise} \end{cases}$$

Now we can define a set of plays $\sigma_g \subseteq M^*_{A[\boldsymbol{\mathcal{U}}]}$ inductively as follows:

$$\sigma_g \ = \ \{\varepsilon\} \cup \{sab \mid s \in \sigma_g \ \wedge \ sa \in P_{A[\boldsymbol{\mathcal{U}}]} \ \wedge \ \hat{g}(a) = b\}.$$

For all $\boldsymbol{B} \trianglelefteq \boldsymbol{\mathcal{U}}$, we can define the restriction of σ_g to $\boldsymbol{B}$ by:

$$\sigma_{\boldsymbol{B}} = \{\varepsilon\} \cup \{sab \in \sigma_g \mid sa \in P_{A[\boldsymbol{B}]}\}.$$

(Note that $\sigma_g = \sigma_{\boldsymbol{\mathcal{U}}}$ in this notation.) We say that σ_g is a *generic strategy* for A, and write $\sigma_g : A$, if the following *restriction condition* is satisfied:

- $\sigma_{\boldsymbol{B}} \subseteq P_{A[\boldsymbol{B}]}$ for all $\boldsymbol{B} \trianglelefteq \boldsymbol{\mathcal{U}}$, so that the restrictions are well-defined.

Note that $\sigma = \sigma_g$ has the following properties.

- σ is a non-empty set of even-length sequences, closed under even-length prefixes.
- σ is *deterministic*, meaning that

$$sab \in \sigma \ \wedge \ sac \in \sigma \ \ \Rightarrow \ \ b = c.$$

- σ is *history-free*, meaning that

$$sab \in \sigma \ \wedge \ t \in \sigma \ \wedge \ ta \in P_{A[\boldsymbol{\mathcal{U}}]} \ \ \Rightarrow \ \ tab \in \sigma.$$

- σ is *generic*:

$$s{\cdot}m_1[m'_1]{\cdot}m_2[m'_2] \in \sigma \wedge t \in \sigma \wedge t{\cdot}m_1[m''_1] \in P_{A[\boldsymbol{\mathcal{U}}]} \ \Rightarrow \ t{\cdot}m_1[m''_1]{\cdot}m_2[m''_1] \in \sigma.$$

These conditions imply that

$$s \cdot m_1[m_1'] \cdot m_2[m_2'] \in \sigma \;\Rightarrow\; m_1' = m_2').$$

Moreover, for any set $\sigma \subseteq P_{A[\mathcal{U}]}$ satisfying the above conditions, there is a least partial function $g : \mathcal{O}_A \rightharpoonup \mathcal{O}_A$ such that $\sigma = \sigma_g$. This function can be defined explicitly by

$$g(m_1) = m_2 \iff \exists s.\, s \cdot m_1[a] \cdot m_2[a] \in \sigma.$$

The equivalence $\approx_A$ on plays can be lifted to a partial equivalence (*i.e.* a symmetric and transitive relation) on strategies on A, which we also write as $\approx$. This is defined most conveniently in terms of a partial pre-order (transitive relation) $\lessapprox$, which is defined as follows.

$$\sigma \lessapprox \tau \;\equiv\; sab \in \sigma \wedge t \in \tau \wedge sa \approx_A ta' \implies \exists b'.\, ta'b' \in \tau \wedge sab \approx_A ta'b'.$$

We can then define

$$\sigma \approx \tau \;\equiv\; \sigma \lessapprox \tau \wedge \tau \lessapprox \sigma.$$

A basic well-formedness condition on strategies σ is that they satisfy this relation, meaning $\sigma \approx \sigma$. Note that for a generic strategy $\sigma = \sigma_{\mathcal{U}}$, using the equivalence on plays in $A[\mathcal{U}]$:

$$\sigma \approx \sigma \implies \sigma_{\boldsymbol{B}} \approx \sigma_{\boldsymbol{B}} \quad \text{for all } \boldsymbol{B} \trianglelefteq \mathcal{U}.$$

A cartesian closed category of games is constructed by taking *partial equivalence classes* of strategies, *i.e.* strategies modulo $\approx$, as morphisms. See [5] for details.

5.1 Copy-Cat Strategies

One additional property of strategies will be important for our purposes. A partial function $f : X \rightharpoonup X$ is said to be a *partial involution* if it is symmetric, *i.e.* if

$$f(x) = y \iff f(y) = x.$$

It is *fixed-point free* if we never have $f(x) = x$. Note that fixed-point free partial involutions on a set X are in bijective correspondence with pairwise disjoint families $\{x_i, y_i\}_{i \in I}$ of two-element subsets of X (*i.e.* the set of pairs $\{x, y\}$ such that $f(x) = y$, and hence also $f(y) = x$). Thus they can thought of as "abstract systems of axiom links". See [6,7] where a combinatory algebra of partial involutions is introduced, and an extensive study is made of realizability over this combinatory algebra.

For us, the important correspondence is with *copy-cat strategies*, first identified in [3] as central to the game-semantical analysis of proofs (and so-named there). We say that σ is a *copy-cat strategy* if $\sigma = \sigma_g$ where g is a fixed-point free partial involution.

Lemma 1 (The Copy-Cat Lemma). *Let $\sigma_g : A$ be a generic copy-cat strategy. If $g(m) = m'$, then for all $s \in \sigma$:*

$$s\lceil m \;=\; s\lceil m'.$$

6 The Model

We shall use the hyperdoctrine formulation of model of System F, as originally proposed by Seeley [22] based on Lawvere's notion of hyperdoctrines [14], and simplified by Pitts [18].

We begin by defining:

$$\mathcal{G}_{\mathcal{U}}(k) \;=\; \mathsf{Sub}(\mathcal{U}) \cap \mathcal{G}(k),$$

where $\mathcal{U}$ is the universe of System F types constructed in Section 6.

6.1 The Base Category

We firstly define a base category $\mathbb{B}$. The objects are natural numbers. A morphism $n \longrightarrow m$ is an m-tuple

$$\langle A_1, \ldots, A_m \rangle, \qquad A_i \in \mathcal{G}_{\mathcal{U}}(n),\ 1 \leq i \leq m.$$

Composition of $\langle A_1, \ldots, A_m \rangle : n \longrightarrow m$ with $\langle B_1, \ldots, A_n \rangle : k \longrightarrow n$ is by substitution:

$$\langle A_1, \ldots, A_m \rangle \circ \boldsymbol{B} \;=\; \langle A_1[\boldsymbol{B}], \ldots, A_m[\boldsymbol{B}] \rangle : k \longrightarrow m.$$

The identities are given by:

$$\mathsf{id}_n \;=\; \langle X_1, \ldots, X_n \rangle.$$

Note that variables act as projections:

$$X_i : n \longrightarrow 1$$

and we can define pairing by

$$\langle \boldsymbol{A}, \boldsymbol{B} \rangle \;=\; \langle A_1, \ldots, A_n, B_1, \ldots, B_m \rangle : k \longrightarrow m + n$$

where

$$\langle A_1, \ldots, A_m \rangle : k \longrightarrow n, \qquad \langle B_1, \ldots, B_m \rangle : k \longrightarrow m.$$

Thus this category has finite products, and is generated by the object 1, in the sense that all objects are finite powers of 1.

6.2 The Indexed CCC

Nest, we define a functor

$$\mathcal{C} : \mathbb{B}^{\mathrm{op}} \longrightarrow \mathbf{CCC}$$

where **CCC** is the category of cartesian closed categories with *specified* products and exponentials, and functors preserving this specified structure.

The cartesian closed category $\mathcal{C}(k)$ has as objects $\mathcal{G}_{\mathcal{U}}(k)$. Note that the objects of $\mathcal{C}(k)$ are the morphisms $\mathbb{B}(k, 1)$; this is part of the Seeley-Pitts definition.

The cartesian closed structure at the object level is given by the constructions on variable games which we have already defined: $A \Rightarrow B$, $A \,\&\, B$, $\mathbf{1}$. Note that $\mathcal{G}_{\mathcal{U}}(k)$ is closed under these constructions by Proposition 7.

A morphism $A \longrightarrow B$ in $\mathcal{C}(k)$ is a generic copy-cat strategy $\sigma : A \Rightarrow B$. Recall that this is actually defined at the "global instance" $\mathcal{U}$:

$$\sigma = \sigma_{\mathcal{U}} : (A \Rightarrow B)[\boldsymbol{\mathcal{U}}] \;\; = \;\; A[\boldsymbol{\mathcal{U}}] \Rightarrow B[\boldsymbol{\mathcal{U}}].$$

More precisely, morphisms are partial equivalence classes of strategies modulo $\approx$.

The cartesian closed structure at the level of morphisms is defined exactly as in [5].

Reindexing It remains to describe the functorial action of morphisms in $\mathbb{B}$. For each $\boldsymbol{C} : n \to m$, we must define a cartesian closed functor

$$\boldsymbol{C}^* : \mathcal{C}(m) \longrightarrow \mathcal{C}(n).$$

We define:

$$\boldsymbol{C}^*(A) \;=\; A[\boldsymbol{C}].$$

If $\sigma : A \Rightarrow B$,

$$\boldsymbol{C}^*(\sigma) = \sigma_{\boldsymbol{C}} : (A \Rightarrow B)[\boldsymbol{C}] \;\; = \;\; A[\boldsymbol{C}] \Rightarrow B[\boldsymbol{C}].$$

For functoriality, note that

$$\boldsymbol{C}^*(\sigma) \circ \boldsymbol{C}^*(\tau) = \sigma_{\boldsymbol{C}} \circ \tau_{\boldsymbol{C}} = (\sigma \circ \tau)_{\boldsymbol{C}} = \boldsymbol{C}^*(\sigma \circ \tau).$$

By Proposition 4, $\boldsymbol{C}^*$ preserves the cartesian closed structure.

6.3 Quantifiers as Adjoints

The second-order quantifiers are interpreted as right adjoints to projections. For each n, we have the projection morphism

$$\langle X_1, \ldots, X_n \rangle : n + 1 \longrightarrow n$$

in $\mathbb{B}$. This yields a functor

$$\boldsymbol{X}^* : \mathcal{C}(n) \longrightarrow \mathcal{C}(n + 1).$$

We must specify a right adjoint

$$\Pi_n : \mathcal{C}(n + 1) \longrightarrow \mathcal{C}(n)$$

to this functor. For $A \in \mathcal{G}_{\mathcal{U}}(n+1)$, we define

$$\Pi_n(A) \;=\; \forall X_{n+1}.\, A.$$

To verify the universal property, for each $C \in \mathcal{G}_{\mathcal{U}}(n)$ we must establish a bijection

$$\Lambda : \mathcal{C}(n)(C, \forall X_{n+1}.\, A) \;\stackrel{\cong}{\longrightarrow}\; \mathcal{C}(n+1)(\boldsymbol{X}^*(C), A).$$

Concretely, note firstly that

$$\boldsymbol{X}^*(C) \;=\; C[\boldsymbol{X}] \;=\; C.$$

Next, note that in both hom-sets the strategies are subsets of $P_{C[\boldsymbol{\mathcal{U}}] \Rightarrow A[\boldsymbol{\mathcal{U}}, \mathcal{U}/X_{n+1}]}$. In the case of generic strategies σ into A, these are subject to the constraint of the *restriction condition*: that is, for each instance $\boldsymbol{B}, B$,

$$\sigma_{\boldsymbol{B},B} \subseteq P_{C[\boldsymbol{B}] \Rightarrow A[\boldsymbol{B},B]}.$$

In the case of strategies σ into $\forall X_{n+1}.\, A$, these are subject to the constraint that for each instance $\boldsymbol{B}$,

$$\sigma_{\boldsymbol{B}} \subseteq P_{C[\boldsymbol{B}] \Rightarrow \forall X_{n+1}.\, A[\boldsymbol{B}, X_{n+1}]}.$$

The equivalence of these conditions follows straightforwardly from Proposition 8. This shows that the required correspondence between these hom-sets is simply the identity (which also disposes of the naturality requirements)!

Naturality (Beck-Chevalley) Finally, we must show that the family of right adjoints Π_n form an indexed (or fibred) adjunction. This amounts to the following: for each $\alpha : m \longrightarrow n$ in $\mathbb{B}$, we must show that

$$\alpha^* \circ \Pi_n = \Pi_m \circ (\alpha \times \mathsf{id}_1)^*.$$

Concretely, if $\alpha = \boldsymbol{C}$, we must show that for each $A \in \mathcal{G}_{\mathcal{U}}(n+1)$,

$$(\forall X_{n+1}.\, A)[\boldsymbol{C}] \;=\; \forall X_{m+1}.\, A[\boldsymbol{C}, X_{m+1}].$$

This is Proposition 9.

Remark We are now in a position to understand the logical significance of the relative polymorphic product $\Pi_i(A, B)$. We could define

$$\mathcal{G}_B(k) \;=\; \mathsf{Sub}(B) \,\cap\, \mathcal{G}(k),$$

and obtain an indexed category $\mathcal{C}_B(k)$ based on $\mathcal{G}_B(k)$ instead of $\mathcal{G}_{\mathcal{U}}(k)$. We would still have an adjunction

$$\mathcal{G}(n)(C, \Pi_{n+1}(A, B)) \;\cong\; \mathcal{C}_B(n+1)(\boldsymbol{X}^*(C), A).$$

However, in general B would not have sufficiently strong closure properties to give rise to a model of System F. Obviously, $\mathsf{Sub}(B)$ must be closed under the cartesian closed operations of product and function space. More subtly, $\mathsf{Sub}(B)$ must be closed under the polymorphic product $\Pi_i(-, B)$. (This is, essentially, the "small completeness" issue [13], although our ambient category of games does not have the requisite exactness properties to allow our construction to be internalised in the style of realizability models.[1]) This circularity, which directly reflects the impredicativity of System F, is resolved by the recursive definition of $\mathcal{U}$.

7 Homomorphisms

We shall now view games as *structures*, and introduce a natural notion of homomorphism between games. These will serve as a useful auxiliary tool in obtaining our results on genericity.

A homomorphism $h : A \longrightarrow B$ is a function

$$h : P_A \longrightarrow P_B$$

which is

- *length-preserving*: $|h(s)| = |s|$
- *prefix-preserving*: $s \sqsubseteq t \;\Rightarrow\; h(s) \sqsubseteq h(t)$
- *equivalence-preserving*: $s \approx t \;\Rightarrow\; h(s) \approx h(t)$.

There is an evident category **Games** with variable games as objects, and homomorphisms as arrows.

Lemma 2 (Play Reconstruction Lemma). *Let A, B be variable games. If we are given $s \in P_A$, and for each $m \in \mathcal{O}^i_A$, a play $t_m \in P_B$ with $|t_m| = \mathsf{numoccs}(m, s)$, then there is a unique $u \in P_{A[B/X_i]}$ such that:*

$$u{\restriction}A = s, \qquad u{\restriction}m = t_m \;\; (m \in \mathcal{O}^i_A).$$

This Lemma makes it easy to define a functorial action of variable games on homomorphisms. Let A be a variable game, and $h : B \longrightarrow C$ a homomorphism. We define

$$A(h) : A[B/X_i] \longrightarrow A[C/X_i]$$

by $A(h)(s) = t$, where

$$t{\restriction}A = s{\restriction}A, \qquad t{\restriction}m = h(s{\restriction}m), \;\; (m \in \mathcal{O}^i_A).$$

[1] However, by the result of Pitts [18], *any* hyperdoctrine model can be fully and faithfully embedded in an (intuitionistic) set-theoretic model.

Lemma 3 (Functoriality Lemma). *$A(h)$ is a well-defined homomorphism, and moreover this action is functorial:*

$$A(g \circ h) = A(g) \circ A(h), \qquad A(\mathsf{id}_B) = \mathsf{id}_{A[B/X_i]}.$$

The second important property is that *homomorphisms preserve plays of generic strategies.*

Lemma 4 (Homomorphism Lemma). *Let A be a variable game, $\sigma : A$ a generic strategy, and $h : C \longrightarrow D$ a homomorphism. Then*

$$s \in \sigma_{A[C/X_i]} \implies h(s) \in \sigma_{A[D/X_i]}.$$

8 Genericity

Our aim in this section is to show that there are generic types in our model, and indeed that, in a sense to be made precise, *most types are generic.*

We fix a variable game $A \in \mathcal{G}(1)$. Out aim is to find conditions on variable games B which imply that, for generic strategies $\sigma, \tau : A$:

$$\sigma_B \approx \tau_B \implies \Lambda(\sigma) \approx \Lambda(\tau) : \forall X.\, A.$$

Since, as explained in Section 8.3,

$$\Lambda(\sigma) \;=\; \sigma \;=\; \sigma_{\mathcal{U}},$$

this reduces to proving the implication

$$\sigma_B \approx \tau_B \implies \sigma_{\mathcal{U}} \approx \tau_{\mathcal{U}}.$$

Our basic result is the following.

Lemma 5 (Genericity Lemma). *If there is a homomorphism $h : \mathcal{U} \longrightarrow B$, then B is generic.*

Remark The Genericity Lemma applies to *any* variable type A; in particular, it is *not* required that A be a sub-game of $\mathcal{U}$. Thus our analysis of genericity is quite robust, and in particular is not limited to System F.

We define the *infinite plays* over a game A as follows: $s \in P_A^\infty$ if every finite prefix of s is in P_A. We can use this notion to give a simple sufficient condition for the hypothesis of the Genericity Lemma to hold.

Lemma 6. *If $P_B^\infty \neq \varnothing$, then B is generic.*

We now apply these ideas to the denotations of System F types, the objective being to show that "most" System F types denote generic instances in the model. Firstly, we define a notion of *length* for games, which we then transfer to types via their denotations as games.

We define

$$|A| \;=\; \sup\{|s| \mid s \in P_A\}.$$

Note that $|A| \leq \omega$.

We now show that any System F type whose denotation admits plays of length greater than 2 is in fact generic!

Lemma 7 (One, Two, Infinity Lemma). *If $|T| > 2$, then T is generic.*

We now give explicit syntactic conditions on System F types which imply that they are generic.

Proposition 10. *Let $T = \forall \boldsymbol{X}. T_1 \to \cdots \to T_k \to X$.*

1. *If for some $i : 1 \leq i \leq k$, $T_i = \forall \boldsymbol{Y}. U_1 \to \cdots \to U_l \to X$, then T is generic.*
2. *If for some $i : 1 \leq i \leq k$, $T_i = \forall \boldsymbol{Y}. U_1 \to \cdots \to U_l \to Y$, and for some $j : 1 \leq j \leq l$, $U_j = \forall \boldsymbol{Z}. V_1 \to \cdots \to V_m \to W$, where W is **either** some $Z_p \in \boldsymbol{Z}$, **or** Y, **or** some $X_q \in \boldsymbol{X}$, then T is generic.*

We apply this to the simple and familiar case of "ML types".

Corollary 1. *Let $T = \forall X. U$, where U is built from the type variable X and $\to$. If U is non-trivial (i.e. it is not just X), then T is generic.*

Examples The following are all examples of generic types.

- $\forall X. X \to X$
- $\forall X. (X \to X) \to X$
- $\forall X. (\forall Y. Y \to Y \to Y) \to X$.

Non-examples The following illustrate the (rather pathological) types which do not fall under the scope of the above results. Note that the first two both have length 1; while the third has length 2.

- $\forall X. X$
- $\forall X. \forall Y. X \to Y$.
- $\forall X. X \to \forall X. X$

Remark An interesting point illustrated by these examples is that our conditions on types are orthogonal to the issue of whether the types are inhabited in System F. Thus the type $\forall X. (X \to X) \to X$ is not inhabited in System F, but is generic in the games model, while the type $\forall X. X \to \forall X. X$ is inhabited in System F, but does not satisfy our conditions for genericity.

9 Related Work

A game semantics for System F was developed by Dominic Hughes in his D.Phil. thesis [12]. A common feature of his approach with ours' is that both give a direct interpretation of open types as certain games, and of type substitution as an operation on games. However, his approach is in a sense rather closer to syntax; it involves carrying type information in the moves, and the resulting model is much more complex. For example, showing that strategies in the model are closed under composition is a major undertaking. Moreover, the main result in [12] is a full completeness theorem essentially stating that the model is isomorphic to the term model of System F (with $\beta\eta$-equivalence), modulo types being reduced to their normal forms. As observed by Longo [16], the term model of System F *does not satisfy Genericity*; in fact, it does not satisfy Axiom (C). It seems that the presence of explicit type information in the moves will preclude the model in [12] from having genericity properties comparable to those we have established for our model.

References

1. M. Abadi, L. Cardelli and P.-L. Curien. Formal Parametric Polymorphism. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, 1993.
2. S. Abramsky. Semantics of Interaction. In *Semantics and Logics of Computation*, edited by A. Pitts and P. Dybjer, Cambridge University Press 1997, 1–32.
3. S. Abramsky, R. Jagadeesan. Games and Full Completeness for Multiplicative Linear Logic, *J. of Symbolic Logic* **59**(2), 1994, 543–574.
4. S. Abramsky, R. Jagadeesan. A Game Semantics for Generic Polymorphism. Oxford University Computing Laboratory Programming Research Group, Research Report RR-03-02, 2003. Available on-line at `http://web.comlab.ox.ac.uk/oucl/publications/tr/rr-03-02`.
5. S. Abramsky, R. Jagadeesan, P. Malacaria. Full Abstraction for PCF, *Inf. and Comp.* **163**, 2000, 409–470.
6. S. Abramsky, M. Lenisa. A Fully-complete PER Model for ML Polymorphic Types, *CSL'00* Conf. Proc., P. Clote, H.Schwichtenberg eds., LNCS **1862**, 2000, 140–155.
7. S. Abramsky, M. Lenisa. A Fully Complete Minimal PER Model for the Simply Typed λ-calculus, *CSL'01* Conf. Proc., LNCS 2001.
8. S. Abramsky and G. McCusker. Games for Recursive Types. In C. Hankin, I. Mackie and R. Nagarajan, eds. *Theory and Formal Methods of Computing 1994*. Imperial College Press, 1995.
9. G. Berry. The Foundations of Esterel. In *Proof, Language and Interaction: Essays in honour of Robin Milner*, eds. G. Plotkin, C. Stirling and M. Tofte. MIT Press 2000, 425–454.
10. J.Y. Girard. Interprétation functionelle et élimunation des coupures de l'arithmètique d'ordre supérieur, Thèse d'Etat, Université Paris VII, 1972.
11. J.-Y. Girard, Y. Lafont and P. Taylor. *Proofs and Types*. Cambridge University Press 1989.
12. D. J. D. Hughes. *Hypergame Semantics: Full Completeness for System F*. D.Phil. thesis, University of Oxford, 1999.

13. J. M. E. Hyland. A small complete category. *Annals of Pure and Applied Logic*, 40, 1988.
14. F. W. Lawvere. Equality in hyperdoctrines and the comprehension schema as an adjoint functor, Proc. Symp. on Applications of Categorical Logic, 1970.
15. G. Longo, K. Milsted, S. Soloviev. The Genericity Theorem and Parametricity in the Polymorphic λ-Calculus, TCS 121(1&2):323–349, 1993.
16. G. Longo. Parametric and Type-Dependent Polymorphism. *Fundamenta Informaticae* 22(1/2):69–92.
17. Q.-Q. Ma and J. C. Reynolds. Types, Abstraction and Parametric Polymorphism, Part 2. In S. Brookes et al. editors, *Mathematical Foundations of Programming Language Semantics*. LNCS **598**, 1992.
18. A. Pitts. Polymorphism is set-theoretic constructively, *CTCS'88* Conf. Proc., D.Pitt ed., LNCS **283**, 1988.
19. G. Plotkin, M. Abadi. A Logic for Parametric Polymorphism, *TLCA'93* Conf. Proc., LNCS, 1993.
20. J. C. Reynolds. Towards a Theory of Type Structure. Programming Symposium, Proceedings, Paris 1974. LNCS **19**, 1974.
21. J. C. Reynolds. Types, Abstraction and Parametric Polymorphism. *Information Processing 83*, pp. 513–523, Elsevier (North-Holland), 1983.
22. R. A. G. Seeley. Categorical semantics for higher-order polymorphic lambda calculus. *Journal of Symbolic Logic*, 52(4):969–989, 1987.

Categories of Containers

Michael Abbott[1], Thorsten Altenkirch[2], and Neil Ghani[1]

[1] Department of Mathematics and Computer Science, University of Leicester
michael@araneidae.co.uk, ng13@mcs.le.ac.uk
[2] School of Computer Science and Information Technology, Nottingham University
txa@cs.nott.ac.uk

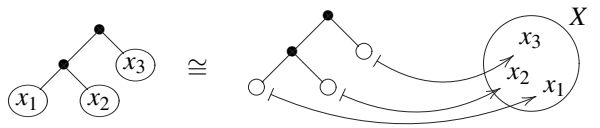
Abstract. We introduce the notion of containers as a mathematical formalisation of the idea that many important datatypes consist of templates where data is stored. We show that containers have good closure properties under a variety of constructions including the formation of initial algebras and final coalgebras. We also show that containers include strictly positive types and shapely types but that there are containers which do not correspond to either of these. Further, we derive a representation result classifying the nature of polymorphic functions between containers. We finish this paper with an application to the theory of shapely types and refer to a forthcoming paper which applies this theory to differentiable types.

1 Introduction

Any element of the type $\mathsf{List}(X)$ of lists over X can be uniquely written as a natural number n given by the length of the list, together with a function $\{1,\ldots,n\} \to X$ which labels each position within the list with an element from X:

$$n : \mathbb{N} \ , \qquad \sigma : \{1\,..\,n\} \to X \ .$$

Similarly, any binary tree tree can be described by its underlying shape which is obtained by deleting the data stored at the leaves with a function mapping the positions in this shape to the data thus:



More generally, we are led to consider datatypes which are given by a set of shapes S and, for each $s \in S$, a family of positions $P(s)$. This presentation of the datatype defines an endofunctor $X \mapsto \coprod_{s\in S} X^{P(s)}$ on **Set**. In this paper we formalise these intuitions by considering families of objects in a locally cartesian closed category $\mathbb{C}$, where the family $s : S \vdash P(s)$ is represented by an object $P \in \mathbb{C}/S$, and the associated functor $T_{S\triangleright P} : \mathbb{C} \to \mathbb{C}$ is defined by $T_{S\triangleright P}X \equiv \Sigma s : S.\,(P(s) \Rightarrow X)$.

We begin by constructing a category $\mathscr{G}$ of "containers", ie syntactic presentations of shapes and positions, and define a full and faithful functor T to the category of endofunctors of $\mathbb{C}$. Given that polymorphic functions are natural transformations, full

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 23–38, 2003.

and faithfulness allows us to classify polymorphic functions between container functors in terms of their action on the shapes and positions of the underlying containers.

We show that $\mathscr{G}$ is complete and cocomplete and that limits and coproducts are preserved by T. This immediately shows that i) container types are closed under products, coproducts and subset types; and ii) this semantics is compositional in that the semantics of a datatype is constructed canonically from the semantics of its parts. The construction of initial algebras and final coalgebras of containers requires, firstly, the definition of containers with multiple parameters and, secondly, a detailed analysis of when T preserves limits and colimits of certain filtered diagrams.

We conclude the paper by relating containers to the shapely types of Jay and Cockett (1994) and Jay (1995). The definition of shapely types does not require the hypothesis of local cartesian closure which we assume, but when $\mathbb{C}$ *is* locally cartesian closed then it turns out that the shapely types are precisely the functors generated by the "discretely finite" containers. A container is discretely finite precisely when each of its objects of positions is locally isomorphic to a finite cardinal.

Further, we gain much by the introduction of extra categorical structure, e.g. the ability to form initial algebras and final coalgebras of containers and the representation result concerning natural transformations between containers. Unlike containers, shapely types are *not* closed under the construction of coinductive types, since the position object of an infinite list cannot be discretely finite.

In this paper we assume that $\mathbb{C}$ is locally finitely presentable (lfp), hence complete and cocomplete, which excludes several interesting examples including Scott domains and realisability models. Here we use the lfp structure for the construction of initial algebras and final coalgebras. In future work we expect to replace this assumption with a more delicate treatment of induction using *internal* structure.

Another application of containers is as a foundation for generic programming within a dependently typed programming framework (Altenkirch and McBride, 2002). An instance of this theme, the derivatives of functors as suggested in McBride (2001), is developed in Abbott et al. (2003) using the material presented here.

The use of the word *container* to refer to a class of datatypes can be found in Hoogendijk and de Moor (2000) who investigated them in a relational setting: their containers are actually closed under quotienting. Containers as introduced here are closely related to analytical functors, which were introduced by Joyal, see Hasegawa (2002). Here we consider them in a more general setting by looking at locally cartesian categories with some additional properties. In the case of **Set** containers are a generalisation of normal functors, closing them under quotients would generalise analytical functors.

In summary, this paper makes the following contributions:

- We develop a new and generic concept of what a container is which is applicable to a wide range of semantic domains.
- We give a representation theorem (Theorem 3.4) which provides a simple analysis of polymorphic functions between datatypes.
- We show a number of closure properties of the category of containers which allow us to interpret all strictly positive types by containers.

– We lay the foundation for a theory of generic programming; a first application is the theory of differentiable datatypes as presented in Abbott et al. (2003).
– We show that Jay and Cockett's shapely types are all containers.

2 Definitions and Notation

This paper implicitly uses the machinery of fibrations (Jacobs 1999, Borceux 1994, chapter 8, etc) to develop the key properties of container categories, and in particular the fullness of the functor T relies on the use of fibred natural transformations. This section collects together the key definitions and results required in this paper.

Given a category with finite limits $\mathbb{C}$, refer to the slice category $\mathbb{C}/A$ over $A \in \mathbb{C}$ as the *fibre* of $\mathbb{C}$ over A. Pullbacks in $\mathbb{C}$ allow us to lift each $f : A \to B$ in $\mathbb{C}$ to a *pullback* or *reindexing* functor $f^* : \mathbb{C}/B \to \mathbb{C}/A$. Assigning a fibre category to each object of $\mathbb{C}$ and a reindexing functor to each morphism of $\mathbb{C}$ is (subject to certain coherence equations) a presentation of a *fibration over* $\mathbb{C}$.

Composition with f yields a functor $\Sigma_f : \mathbb{C}/A \to \mathbb{C}/B$ left adjoint to f^*. $\mathbb{C}$ is *locally cartesian closed* iff each fibre of $\mathbb{C}$ is cartesian closed, or equivalently, if each pullback functor f^* has a right adjoint $f^* \dashv \Pi_f$.

Each exponential category $\mathbb{C}^I$ can in turn be regarded as fibred over $\mathbb{C}$ by taking the fibre of $\mathbb{C}^I$ over $A \in \mathbb{C}$ equal to $(\mathbb{C}/A)^I$. Now define $[\mathbb{C}^I, \mathbb{C}^J]$ to be the category of fibred functors $F : \mathbb{C}^I \to \mathbb{C}^J$ and fibred natural transformations, where each F is a family $F_A : (\mathbb{C}/A)^I \to (\mathbb{C}/A)^J$ such that $(f^*)^J F_B \cong F_A (f^*)^I$ for each $f : A \to B$ and similarly for natural transformations.

Write $a : A \vdash B(a)$ or even just $A \vdash B$ for $B \in \mathbb{C}/A$. We'll write $a : A, b : B(a) \vdash C(a,b)$ as a shorthand for $(a,b) : \Sigma_A B \vdash C(a,b)$. When dealing with a collection A_i for $i \in I$, we'll write this as $(A_i)_{i \in I}$ or $\vec{A}$ or even just A. Write $\Sigma a : A$ and $\Pi a : A$ for the Σ and Π types corresponding to the adjoints to reindexing. Substitution in variables will be used interchangeably with substitution by pullback, so $A \vdash f^* B$ may also be written as $a : A \vdash B(f(a))$ or $a : A \vdash B(fa)$. The signs $\coprod$ and $\prod$ will be used for coproducts and products respectively over external sets, while Σ and Π refer to the corresponding internal constructions in $\mathbb{C}$. See Hofmann (1997) for a more detailed explanation of the interaction between type theory and semantics assumed in this paper.

Limits and colimits are *fibred* iff they exist in each fibre and are preserved by reindexing functors. Limits and colimits in a locally cartesian closed category $\mathbb{C}$ are automatically fibred. This useful result allows us to omit the qualification that limits and colimits be "fibred" throughout this paper.

When $\mathbb{C}$ is locally cartesian closed say that coproducts are *disjoint* (or equivalently that $\mathbb{C}$ is *extensive*)[1] iff the pullback of distinct coprojections $\kappa_i : A_i \to \coprod_{i \in I} A_i$ into a coproduct is always the initial object 0. Henceforth, we'll assume that $\mathbb{C}$ has finite limits, is locally cartesian closed and has disjoint coproducts. The following notion of "disjoint fibres" follows from disjoint coproducts.

[1] For general $\mathbb{C}$, coproducts are disjoint iff coprojections are also mono, and $\mathbb{C}$ is extensive iff coproducts are disjoint and are preserved by pullbacks.

Proposition 2.1. *If $\mathbb{C}$ has disjoint coproducts then the functor $\vec{\kappa}^* : \mathbb{C}/\coprod_{i\in I} A_i \to \prod_{i\in I}(\mathbb{C}/A_i)$, taking $\coprod_{i\in I} A_i \vdash B$ to $(A_i \vdash \kappa_i^* B)_{i\in I}$, is an equivalence. Say that $\mathbb{C}$ has* disjoint fibres *when this holds.* □

Write $\bigoplus : \prod_{i\in I}(\mathbb{C}/A_i) \to \mathbb{C}/\coprod_{i\in I} A_i$ for the adjoint to $\vec{\kappa}^*$ and $-\dagger-$ for the binary case. Note that $\bigoplus_{i\in I} B_i \cong \coprod_{i\in I} \Sigma_{\kappa_i} B_i$ for $(A_i \vdash B_i)_{i\in I} \in \prod_{i\in I}(\mathbb{C}/A_i)$.

The following lemma collects together some useful identities which hold in any category considered in this paper.

Lemma 2.2. *For extensive locally cartesian closed $\mathbb{C}$ the following isomorphisms hold (*IC *stands for* intensional choice*,* Cu *for* Curry *and* DF *for* disjoint fibres*):*

$$\Pi a : A.\, \Sigma b : B(a).\, C(a,b) \cong \Sigma f : (\Pi a : A.\, B(a)).\, \Pi a : A.\, C(a, fa) \tag{IC1}$$

$$\prod_{i\in I} \Sigma b : B_i.\, C_i(b) \cong \Sigma a : \prod_{i\in I} B_i.\, \prod_{i\in I} C_i(\pi_i a) \tag{IC2}$$

$$\Pi a : A.\, (B(a) \Rightarrow C) \cong (\Sigma a : A.\, B(a)) \Rightarrow C \tag{Cu1}$$

$$\prod_{i\in I} (B_i \Rightarrow C) \cong \left(\coprod_{i\in I} B_i\right) \Rightarrow C \tag{Cu2}$$

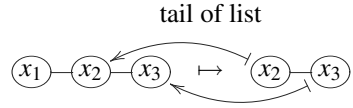
$$\left(\bigoplus_{i\in I} B_i\right)(\kappa_i a) \cong B_i(a) \tag{DF1}$$

$$\coprod_{i\in I} \Sigma a : A_i.\, C(\kappa_i a) \cong \Sigma a : \coprod_{i\in I} A_i.\, C(a) \tag{DF2}$$ □

For technical convenience, a choice of pullbacks is assumed in $\mathbb{C}$ (this ensures that our fibrations are cloven). Finally, note that we make essential use of classical set theory with choice in the meta-theory in theorem 5.6 and proposition 6.6. It should be possible to avoid this dependency by developing more of the theory internally to $\mathbb{C}$.

3 Basic Properties of Containers

The basic notion of a *container* is a dependent pair of types $A \vdash B$ creating a functor $T_{A\triangleright B} X \equiv \Sigma a : A.\, (B(a) \Rightarrow X)$. In order to understand a morphism of containers, consider the map $\mathsf{tail} : \mathsf{List}\, X \to 1 + \mathsf{List}\, X$ taking the empty list to 1 and otherwise yielding the tail of the given list:



This map is defined by i) a choice of shape in $1 + \mathsf{List}\, X$ for each shape in $\mathsf{List}\, X$; and ii) for each position in the chosen shape a position in the original shape. Thus a morphism of containers $(A \vdash B) \to (C \vdash D)$ is a pair of morphisms $(u : A \to C,\, f : u^* D \to B)$. With this definition of a category $\mathscr{G}$ of containers we can construct a full and faithful functor $T : \mathscr{G} \to [\mathbb{C}, \mathbb{C}]$ and show the completeness properties discussed in the introduction.

However, when constructing fixed points it is also necessary to take account of containers with parameters, so we define $T : \mathscr{G}_I \to [\mathbb{C}^I, \mathbb{C}]$ for each parameter index set I. For the purposes of this paper the index set n or I will generally be a finite set, but this

makes little difference. Indeed, it is straightforward to generalise the development in this paper to the case where containers are parameterised by *internal* index objects $I \in \mathbb{C}$; when $\mathbb{C}$ has enough coproducts nothing is lost by doing this, since $\mathbb{C}^I \simeq \mathbb{C}/\coprod_{i\in I} 1$. This generalisation will be important for future developments of this theory, but is not required in this paper.

Definition 3.1. *Given an index set I define the* category of containers $\mathscr{G}_I$ *as follows:*

- *Objects are pairs $(A \in \mathbb{C}, B \in (\mathbb{C}/A)^I)$; write this as $(A \triangleright B) \in \mathscr{G}_I$*
- *A morphism $(A \triangleright B) \to (C \triangleright D)$ is a pair (u, f) for $u : A \to C$ in $\mathbb{C}$ and $f : (u^*)^I D \to B$ in $(\mathbb{C}/A)^I$.*

A container $(A \triangleright B) \in \mathscr{G}_I$ can be written using type theoretic notation as

$$\vdash A \qquad i : I, a : A \vdash B_i(a) \ .$$

A morphism $(u, f) : (A \triangleright B) \to (C \triangleright D)$ can be written in type theoretic notation as

$$u : A \to C \qquad i : I, a : A \vdash f_i(a) : D_i(ua) \to B_i(a) \ .$$

Finally, each $(A \triangleright B) \in \mathscr{G}_I$, thought of as a syntactic presentation of a datatype, generates a fibred functor $T_{A\triangleright B} : \mathbb{C}^I \to \mathbb{C}$ which is its semantics.

Definition 3.2. *Define the* container construction functor $T : \mathscr{G}_I \to [\mathbb{C}^I, \mathbb{C}]$ *as follows. Given $(A \triangleright B) \in \mathscr{G}_I$ and $X \in \mathbb{C}^I$ define*

$$T_{A\triangleright B} X \equiv \Sigma a : A. \prod_{i\in I} (B_i(a) \Rightarrow X_i) \ ,$$

and for $(u, f) : (A \triangleright B) \to (C \triangleright D)$ define $T_{u,f} : T_{A\triangleright B} \to T_{C\triangleright D}$ to be the natural transformation $T_{u,f} X : T_{A\triangleright B} X \to T_{C\triangleright D} X$ thus:

$$(a, g) : T_{A\triangleright B} X \vdash T_{u,f} X(a, g) \equiv (u(a), (g_i \cdot f_i)_{i\in I}) \ .$$

The following proposition follows more or less immediately by the construction of T.

Proposition 3.3. *For each container $F \in \mathscr{G}_I$ and each container morphism $\alpha : F \to G$ the functor T_F and natural transformation T_α are fibred over $\mathbb{C}$.* □

By making essential use of the fact that the natural transformations in $[\mathbb{C}^I, \mathbb{C}]$ are *fibred* (c.f. section 2) we can show that T is full and faithful.

Theorem 3.4. *The functor $T : \mathscr{G}_I \to [\mathbb{C}^I, \mathbb{C}]$ is full and faithful.*

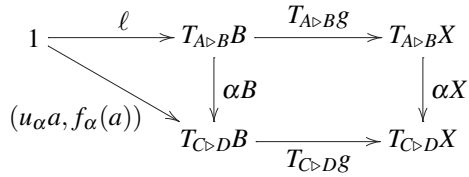
Proof. To show that T is full and faithful it is sufficient to lift each natural transformation $\alpha : T_{A\triangleright B} \to T_{C\triangleright D}$ in $[\mathbb{C}^I, \mathbb{C}]$ to a map $(u_\alpha, f_\alpha) : (A \triangleright B) \to (C \triangleright D)$ in $\mathscr{G}_I$ and show this construction is inverse to T.

Given $\alpha : T_{A\triangleright B} \to T_{C\triangleright D}$ construct $\ell \equiv (a', \mathrm{id}_{B(a')}) \in T_{A\triangleright B} B$ in the fibre $\mathbb{C}/A$ (or in terms of type theory, add $a' : A$ to the context). We can now construct $\alpha B \cdot \ell \in T_{C\triangleright D} B = \Sigma c : C. \prod_{i\in I} (D_i(c) \Rightarrow B_i(a'))$ in the same context, and write $\alpha B \cdot \ell = (u_\alpha, f_\alpha)$ where $u_\alpha(a') : C$ and $f_\alpha(a') : \prod_{i\in I} (D_i(u_\alpha a') \Rightarrow B_i(a'))$ for $a' : A$.

Thus (u_α, f_α) can be understood as a morphism $(A \triangleright B) \to (C \triangleright D)$ in $\mathscr{G}_I$. It remains to show that this construction is inverse to T.

When $\alpha = T_{u,f}$, just evaluate $\alpha B \cdot \ell = (ua', \mathrm{id} \cdot f)$, which corresponds to the original map (u, f).

To show in general that $\alpha = T_{u_\alpha, f_\alpha}$, let $X \in \mathbb{C}^I$, $a : A$ and $g : \prod_{i \in I}(B_i(a) \Rightarrow X_i)$ be given, consider the diagram



and evaluate

$$\begin{aligned} \alpha X \cdot (a, g) &= \alpha X \cdot T_{A \triangleright B} g \cdot \ell = T_{C \triangleright D} g \cdot \alpha B \cdot \ell = T_{C \triangleright D} g \cdot (u_\alpha a, f_\alpha(a)) \\ &= (u_\alpha a, g \cdot f_\alpha(a)) = T_{u_\alpha, f_\alpha} X \cdot (a, g) \quad . \end{aligned}$$

This shows that $\alpha = T_{u_\alpha, f_\alpha}$ as required. □

This theorem gives a particularly simple analysis of polymorphic functions between container functors. For example, it is easy to observe that there are precisely n^m polymorphic functions $X^n \to X^m$: the data type X^n is the container $(1 \triangleright n)$ and hence there is a bijection between polymorphic functions $X^n \to X^m$ and functions $m \to n$. Similarly, any polymorphic function $\mathsf{List} X \to \mathsf{List} X$ can be uniquely written as a function $u : \mathbb{N} \to \mathbb{N}$ together with for each natural number $n : \mathbb{N}$ a function $f_n : un \to n$.

4 Limits and Colimits of Containers

It turns out that each $\mathscr{G}_I$ inherits completeness and cocompleteness from $\mathbb{C}$, and that T preserves completeness. Preservation of cocompleteness is more complex, and only a limited class of colimits are preserved by T.

Proposition 4.1. *If $\mathbb{C}$ has limits and colimits of shape $\mathbb{J}$ then $\mathscr{G}_I$ has limits of shape $\mathbb{J}$ and T preserves these limits.*

Proof. We'll proceed by appealing to the fact that T reflects limits (since it is full and faithful), and the proof will proceed separately for products and equalisers.

Products. Let $(A_k \triangleright B_k)_{k \in K}$ be a family of objects in $\mathscr{G}_I$ and compute (the labels refer to lemma 2.2)

$$\begin{aligned} \textstyle\prod_{k \in K} T_{A_k \triangleright B_k} X &= \textstyle\prod_{k \in K} \Sigma a : A. \prod_{i \in I} (B_{k,i}(a) \Rightarrow X_i) \\ &\cong \Sigma a : \textstyle\prod_{k \in K} A_k. \prod_{k \in K} \prod_{i \in I} (B_{k,i}(\pi_k a) \Rightarrow X_i) && \text{(IC2)} \\ &\cong \Sigma a : \textstyle\prod_{k \in K} A_k. \prod_{i \in I} \left(\left(\coprod_{k \in K} B_{k,i}(\pi_k a) \right) \Rightarrow X_i \right) && \text{(Cu2)} \\ &= T_{\prod_{k \in K} A_k \triangleright \coprod_{k \in K} (\pi_k^*)^I B_k} X \end{aligned}$$

showing by reflection along T that

$$\prod_{k\in K}(A_k \triangleright B_k) \cong \left(\prod_{k\in K} A_k \;\triangleright\; \coprod_{k\in K}(\pi_k^*)^I B_k\right) \quad .$$

Equalisers. Given parallel maps $(u,f),(v,g):(A\triangleright B)\rightrightarrows(C\triangleright D)$ construct

$$(E\triangleright Q) \rightarrowtail (A\triangleright B) \underset{(v,g)}{\overset{(u,f)}{\rightrightarrows}} (C\triangleright D)$$

where e is the equaliser in $\mathbb{C}$ of u,v and q is the coequaliser in $(\mathbb{C}/E)^I$ of $(e^*)^I f$, $(e^*)^I g$. To show that $T_{e,q}$ is the equaliser of $T_{u,f}, T_{v,g}$ fix $X\in\mathbb{C}^I$, $U\in\mathbb{C}$ and let $\alpha: U\to T_{A\triangleright B}X$ be given equalising this parallel pair at X.

For $x:U$ write $\alpha(x)=(a,h)$ where $a:A$, $h:\prod_{i\in I}(B_i(a)\Rightarrow X_i)$. The condition on α tells us that $u(a)=v(a)$ and so there is a unique $y:E$ with $a=e(y)$. Similarly we know that $h\cdot f(ey)=h\cdot g(ey)$ and in particular there is a unique $k:Q(y)\to X$ with $h=k\cdot q$.

The assignment $x\mapsto(y,k)$ defines a map $\beta: U\to T_{E\triangleright Q}X$ giving a unique factorisation of α, showing that $T_{e,q}X$ is an equaliser and hence so is (e,q). □

In particular, this result tells us that the limit in $[\mathbb{C}^I,\mathbb{C}]$ of a diagram of container functors is itself a container functor.

It's nice to see that coproducts of containers are also well behaved.

Proposition 4.2. *If $\mathbb{C}$ has products and coproducts of size K then $\mathscr{G}_I$ has coproducts of size K preserved by T.*

Proof. Given a family $(A_k\vdash B_k)_{k\in K}$ of objects in $\mathscr{G}_I$ calculate (making essential use of disjointness of fibres):

$$\begin{aligned}
\coprod_{k\in K} T_{A_k\triangleright B_k}X &= \coprod_{k\in K}\Sigma a:A_k.\prod_{i\in I}(B_{k,i}(a)\Rightarrow X_i)\\
&\cong \coprod_{k\in K}\Sigma a:A_k.\prod_{i\in I}\Big((\coprod_{k'\in K} B_{k',i})(\kappa_k a)\Rightarrow X_i\Big) && \text{(DF1)}\\
&\cong \Sigma a:\coprod_{k\in K}A_k.\prod_{i\in I}\Big((\coprod_{k\in K} B_{k,i})(a)\Rightarrow X_i\Big) && \text{(DF2)}\\
&= T_{\coprod_{k\in K}A_k\triangleright(\coprod_{k\in K}B_{j,i})_{i\in I}}X
\end{aligned}$$

showing by reflection along T that

$$\coprod_{k\in K}(A_k\triangleright B_k)\cong\left(\coprod_{k\in K}A_k\;\triangleright\;\coprod_{k\in K}B_k\right)\quad . \qquad \square$$

The fate of coequalisers is more complicated. It turns out that $\mathscr{G}_I$ has coequalisers when $\mathbb{C}$ has both equalisers and coequalisers, but they are *not* preserved by T.

The following proposition is presented without proof (the construction of coequalisers in $\mathscr{G}$ is fairly complex and is not required in this paper).

Proposition 4.3. *If $\mathbb{C}$ has equalisers and coequalisers then $\mathscr{G}_I$ has coequalisers.* □

The following example shows that coequalisers are not preserved by T.

Example 4.4. Consider the following coequaliser diagram in $[\mathbb{C},\mathbb{C}]$

$$X \times X \underset{(\pi',\pi)}{\overset{\mathrm{id}_{X\times X}}{\rightrightarrows}} X \times X \twoheadrightarrow (X \times X)/\sim$$

where $(x,y) \sim (y,x)$. The functor $X \mapsto X \times X$ is a container functor generated by $(1 \triangleright 2)$, and the coequaliser of the corresponding parallel pair in $\mathscr{G}_1$ is the container $(1 \triangleright 0)$. Note however that $T_{1\triangleright 0}X \cong 1 \ncong (X \times X)/\sim$.

Unfortunately, filtered colimits aren't preserved by T either.

Example 4.5. Consider the ω-chain in $\mathscr{G}_1$ given by $n \mapsto (1 \triangleright A^n)$ (for fixed A) on objects and $(n \to n+m) \mapsto \pi_{n,m} : A^{n+m} \cong A^n \times A^m \to A^n$ on maps. The filtered colimit of this diagram can be computed in $\mathscr{G}_1$ to be $(1 \triangleright A^{\mathbb{N}})$. However, applying T to this diagram produces the ω-chain

$$X \xrightarrow{X^{\pi_{0,1}}} X^A \xrightarrow{X^{\pi_{1,1}}} X^{A^2} \xrightarrow{X^{\pi_{2,1}}} \cdots$$

and the colimit of this chain in **Set** is strictly smaller than $X^{A^{\mathbb{N}}}$.

5 Filtered Colimits of Cartesian Diagrams

Although $\mathscr{G}_I$ has colimits they are not preserved by T, and this also applies to filtered colimits. As we will want to use filtered colimits for the construction of initial algebras, this is a potential problem. Fortunately, there exists a class of filtered colimit diagrams which is both sufficient for the construction of initial algebras and which *are* preserved by T.

Throughout this section take $\mathbb{C}$ to be finitely accessible ($\mathbb{C}$ has filtered colimits and a generating set of finitely presentable objects, Adámek and Rosický 1994) as well as being locally cartesian closed.

Definition 5.1. *A morphism (u,f) in $\mathscr{G}_I$ is* cartesian *iff f is an isomorphism*[2].

For each u there is a bijection between cartesian morphisms $(u,f) : (A \triangleright B) \to (C \triangleright D)$ in $\mathscr{G}_I$ and morphisms $\bar{f}$ in $\mathbb{C}^I$ making each square below a pullback:

$$\begin{array}{ccc} B_i & \xrightarrow{\bar{f}_i} & D_i \\ \downarrow & \lrcorner & \downarrow \\ A & \xrightarrow[u]{} & C \end{array} \quad .$$

[2] (u,f) is *cartesian* with respect to this definition precisely when it is cartesian (in the sense of fibrations) with respect to the projection functor $\pi : \mathscr{G}_I \to \mathbb{C}$ taking $(A \triangleright B)$ to A.

We can also translate the notion of cartesian morphism into natural transformations between container functors: a natural transformation $\alpha : T_{A \triangleright B} \to T_{C \triangleright D}$ derives from a cartesian map iff the naturality squares of α are all pullbacks (such natural transformations are often also called *cartesian*, in this case with respect to the "evaluation at 1" functor).

Define $\widehat{\mathscr{G}_I}$ to have the same objects as $\mathscr{G}_I$ but only cartesian arrows as morphisms. We will show that $\widehat{\mathscr{G}_I}$ has filtered colimits which are preserved by T (when restricted to $\widehat{\mathscr{G}_I}$), and hence also by the inclusion $\widehat{\mathscr{G}_I} \hookrightarrow \mathscr{G}_I$.

The lemma below follows directly from the corresponding result in **Set** and helps us work with maps from finitely presentable objects to filtered colimits (write $\bigvee D$ for the colimit of a filtered diagram D).

Lemma 5.2. *Let $D : \mathbb{J} \to \mathbb{C}$ be a filtered diagram with colimiting cone $d : D \to \bigvee D$ and let U be finitely presentable.*

1. *For each $\alpha : U \to \bigvee D$ there exists $J \in \mathbb{J}$ and $\alpha_J : U \to DJ$ such that $\alpha = d_J \cdot \alpha_J$.*
2. *Given $\alpha : U \to DI$, $\beta : U \to DJ$ such that $d_I \cdot \alpha = d_J \cdot \beta$ there exists $K \in \mathbb{J}$ and maps $f : I \to K$, $g : J \to K$ such that $Df \cdot \alpha = Dg \cdot \beta$.* □

Before the main result we need a technical lemma about filtered colimits in finitely accessible categories.

Lemma 5.3. *Given a filtered diagram in $\mathbb{C}^{\to}$ with every edge a pullback then the arrows of the colimiting cone are also pullbacks.*

Proof. We need to show, for each $I \in \mathbb{C}$, that the square

$$\begin{array}{ccc} EI & \xrightarrow{e_I} & \bigvee E \\ {\scriptstyle \alpha_I}\downarrow & & \downarrow{\scriptstyle \bar{\alpha}} \\ DI & \xrightarrow[d_I]{} & \bigvee D \end{array}$$

is a pullback, where $E \xrightarrow{\alpha} D$ is the diagram, (d,e) are the components of its colimiting cone and $\bar{\alpha}$ is the factorisation of $d \cdot \alpha$ through e. So let a cone $DI \xleftarrow{a} U \xrightarrow{b} \bigvee E$ satisfying $d_I \cdot a = \bar{\alpha} \cdot b$ be given. Without loss of generality we can assume that U is finitely presentable and we can now appeal to lemma 5.2 above.

Construct first $b_J : U \to EJ$ such that $b = e_J \cdot b_J$; then as $d_I \cdot a = \bar{\alpha} \cdot e_J \cdot b_J = d_J \cdot (\alpha_J \cdot b_J)$ there exist $f : I \to K$, $g : J \to K$ with $Df \cdot a = Dg \cdot \alpha_J \cdot b_J = \alpha_K \cdot Eg \cdot b_J$ and so we can construct a factorisation $b_I : U \to EI$ through the pullback over f satisfying $\alpha_I \cdot b_I = a$ and $Ef \cdot b_I = Eg \cdot b_J$. This is a factorisation of (a,b) since $e_I \cdot b_I = e_K \cdot Ef \cdot b_I = e_K \cdot Eg \cdot e_J = e_J \cdot b_J = b$.

This factorisation is unique. Let $b, b' : U \rightrightarrows EI$ be given such that $e_I \cdot b = e_I \cdot b'$. Then there exist $f, f' : I \rightrightarrows J$ with $Ef \cdot b = Ef' \cdot b'$; but indeed there exists $g : J \to K$ with $h \equiv g \cdot f = g \cdot f'$ and so $Eh \cdot b = Eh \cdot b'$. As the square over h is a pullback we can conclude $b = b'$. □

Now we are in a position to state the main result, that the filtered colimit of a cartesian diagram of container functors is itself a container functor.

Proposition 5.4. *For each set I the category $\widehat{\mathscr{G}_I}$ has filtered colimits which are preserved by T.*

Proof. Let a diagram $(D \triangleright E) : \mathbb{J} \to \widehat{\mathscr{G}_I}$ be given, i.e. for each $K \in \mathbb{J}$ there is a container $(DK \triangleright EK)$ and for each $f : K \to L$ a cartesian container morphism (Df, Ef).

For each $f : K \to L$ in $\mathbb{J}$, write $\bar{E}f$ for the map $EK \to EL$ derived from cartesian Ef so that we get the left hand pullback square below:

$$\begin{array}{ccccc} EK & \xrightarrow{\bar{E}f} & EL & \xrightarrow{\bar{e}_L} & \bigvee \bar{E} \\ \downarrow & \lrcorner & \downarrow & \lrcorner & \downarrow \\ DK & \xrightarrow[\bar{D}f]{} & DL & \xrightarrow[d_L]{} & \bigvee D \end{array} .$$

After taking the colimits shown (with colimiting cones d and $\bar{e}$), we know from lemma 5.3 that the right hand square is also a pullback and we can interpret the right hand side as a container together with a cartesian cone $(d, e) : (D \triangleright E) \dot{\to} (\bigvee D \triangleright \bigvee \bar{E})$.

It remains to show that $T_{\bigvee D \triangleright \bigvee \bar{E}} \cong \bigvee T_{D \triangleright E}$, so let a cone $f : T_{D \triangleright E} X \dot{\to} U$ be given as shown below, where the map k_K takes (a, g) to $(d_K(a), g)$, using the isomorphism $(\bigvee \bar{E})_i(d_K(a)) \cong EK_i(a)$ (for $K \in \mathbb{J}$, $i : I$, $a : DK_j$) derived from (d, e) cartesian.

$$\begin{array}{ccc} \Sigma a : DK_j . \prod_{i \in I} (EK_i(a) \Rightarrow X_i) & \xrightarrow{k_K} & \Sigma a : \bigvee D . \prod_{i \in I} ((\bigvee \bar{E})_i(a) \Rightarrow X_i) \\ & {\scriptstyle f_K} \searrow \quad \swarrow {\scriptstyle h} & \\ & U & \end{array}$$

To construct h let $a : \bigvee D$ and $g : \prod_{i \in i} ((\bigvee D)_i(a) \Rightarrow X_i)$ be given and choose $K \in \mathbb{J}$, $a_K \in DK$ such that $a = d_K(a_K)$, and so we have $(a_K, g) : T_{DK \triangleright EK} X$ and can compute $h(a, g) \equiv f_K(a_K, g)$; this construction of $h(a, g)$ is unique and independent of the choice of K and a_K. □

Finally the above proposition can be applied to the construction of fixed points on $\mathscr{G}_I$.

Definition 5.5. *Say that an endofunctor F on a category with filtered colimits* has rank *iff there exists a cardinal $\aleph$, the* rank *of F, such that F preserves $\aleph$-filtered colimits.*

The following theorem is a variant of Adámek and Koubek (1979).

Theorem 5.6 (Adámek). *If a category $\mathbb{C}$ has an initial object and colimits of all filtered diagrams then every endofunctor on $\mathbb{C}$ with rank has an initial algebra.*

If $G : \mathbb{C} \to \mathbb{D}$ preserves the initial object and all filtered colimits then any endofunctor F' on $\mathbb{D}$ satisfying $F'G \cong GF$ for some endofunctor F on $\mathbb{C}$ with rank has an initial algebra given by the image under G of the initial algebra of F. □

The construction of initial algebras in $\mathscr{G}$ now follows as a corollary of the above.

Theorem 5.7. *Let F be an endofunctor on $\mathscr{G}_I$ such that F restricts to an endofunctor $\hat{F}$ on $\widehat{\mathscr{G}_I}$ (i.e., F preserves cartesian morphisms) and such that $\hat{F}$ has rank, then F has an initial algebra $\mu F \in \mathscr{G}_I$ which is preserved by T.*

Proof. We've established that $\widehat{\mathscr{G}_I}$ has filtered colimits which are preserved by $\widehat{\mathscr{G}_I} \hookrightarrow \mathscr{G}_I$ and by T and it's clear that the initial object of $\mathscr{G}_I$ is initial in $\widehat{\mathscr{G}_I}$ and is also preserved by T and so we can apply theorem 5.6. □

As noted in section 2 it would be desirable to have a constructive version of this theorem, probably along the lines suggested by Taylor (1999, Section 6.7).

6 Fixed Points of Containers

Categories of containers are, under suitable assumptions, closed under the operations of taking least and greatest fixed points, or in other words given a container functor $F(\vec{X},Y)$ in $n+1$ parameters the types $\mu Y.F(\vec{X},Y)$ and $\nu Y.F(\vec{X},Y)$ are containers (in n parameters).

The least and greatest fixed points of a type are defined by repeated substitution, for example the type $\nu Y.F(\vec{X},Y)$ can be constructed as the limit of the ω-chain

$$1 \longleftarrow F(\vec{X},1) \longleftarrow F(\vec{X},F(\vec{X},1)) \longleftarrow \cdots \longleftarrow \varprojlim_{n<\omega} F^n[1]$$

where we write $F[Y] \equiv F(\vec{X},Y)$ (note that the ν type only needs ω-limits for its construction, but as discussed below, μ types can require colimits of transfinite chains[3]). Therefore the first thing we need to do is to define the composition of two containers.

Given containers $F \in \mathscr{G}_{I+1}$ and $G \in \mathscr{G}_I$ we can compose their images under T to construct the functor

$$T_F[T_G] \equiv (\mathbb{C}^I \xrightarrow{(\mathrm{id}_{\mathbb{C}^I}, T_G)} \mathbb{C}^I \times \mathbb{C} \cong \mathbb{C}^{I+1} \xrightarrow{T_F} \mathbb{C}) \ .$$

This composition can be lifted to a functor $-[-] : \mathscr{G}_{I+1} \times \mathscr{G}_I \to \mathscr{G}_I$ as follows. For a container in $\mathscr{G}_{I+1}$ write $(A \triangleright B, E) \in \mathscr{G}_{I+1}$, where $B \in (\mathbb{C}/A)^I$ and $E \in \mathbb{C}/A$ and define:

$$(A \triangleright B, E)[(C \triangleright D)] \equiv (a:A,\, f:E(a) \Rightarrow C \ \triangleright \ (B_i(a) + \Sigma e : E(a).\, D_i(fe))_{i \in I}) \ .$$

In other words, given type constructors $F(\vec{X},Y)$ and $G(\vec{X})$ this construction defines the composite type constructor $F[G](\vec{X}) \equiv F(\vec{X}, G(\vec{X}))$.

[3] For example, the type of ω-branching trees, $\mu Y.X + (\mathbb{N} \Rightarrow Y)$, cannot be constructed using only ω-colimits.

Proposition 6.1. *Composition of containers commutes with composition of functors thus:* $T_F[T_G] \cong T_{F[G]}$.

Proof. Calculate (for conciseness we write exponentials using superscripts where convenient and write Σ_A for $\Sigma a : A.$ throughout, eliding the parameter a):

$$\begin{aligned}
T_{A \triangleright B,E}[T_{C \triangleright D}]X &= \Sigma_A \left(\left(\prod_{i \in I} X_i^{B_i} \right) \times \left(E \Rightarrow \Sigma c : C. \prod_{i \in I} X_i^{D_i(c)} \right) \right) \\
&\cong \Sigma_A \left(\left(\prod_{i \in I} X_i^{B_i} \right) \times \left(\Sigma f : C^E . \Pi e : E. \prod_{i \in I} X_i^{D_i(fe)} \right) \right) && \text{(IC1)} \\
&\cong \Sigma_A \Sigma f : C^E . \prod_{i \in I} \left(X_i^{B_i} \times \left(\Pi e : E. X_i^{D_i(fe)} \right) \right) \\
&\cong \Sigma_A \Sigma f : C^E . \prod_{i \in I} \left((B_i + \Sigma e : E. D_i(fe)) \Rightarrow X_i \right) && \text{(Cu1, Cu2)} \\
&\cong T_{(A \triangleright B,E)[C \triangleright D]} X \quad .
\end{aligned}$$

As all the above isomorphisms are natural in X we get the desired isomorphism of functors. □

The next lemma is useful for the construction of both least and greatest fixed points and has other applications. In particular, T_F preserves both pullbacks and cofiltered limits.

Lemma 6.2. *For* $(A \triangleright B) \in \mathscr{G}_I$ *the functor* $T_{A \triangleright B}$ *preserves limits of connected non-empty diagrams (*connected limits*).*

Proof. Since $\prod$ and $\Rightarrow$ preserve limits, it is sufficient to observe that Σ_A preserves connected limits, which is noted, for example, in Carboni and Johnstone (1995). □

Corollary 6.3. *For each* $F \in \mathscr{G}_{I+1}$ *the functor* $F[-] : \mathscr{G}_I \to \mathscr{G}_I$ *preserves connected limits.*

Proof. Let D be a non-empty connected diagram, then since T_F preserves connected limits it is easy to see that $T_F[\varprojlim D] \cong \varprojlim (T_F[D])$. Since T preserves limits we can calculate

$$T_{F[\varprojlim D]} \cong T_F[T_{\varprojlim D}] \cong T_F[\varprojlim T_D] \cong \varprojlim (T_F[T_D]) \cong \varprojlim T_{F[D]} \cong T_{\varprojlim (F[D])}$$

and so by reflection along T conclude that $F[\varprojlim D] \cong \varprojlim (F[D])$. □

We can immediately conclude that if $\mathbb{C}$ is complete and cocomplete (in fact, ω- limits and colimits are sufficient) then containers have final coalgebras.

Theorem 6.4. *Each* $F \in \mathscr{G}_{I+1}$ *has a final coalgebra* $\nu F \in \mathscr{G}_I$ *which is preserved by* T *(and so satisfies* $T_{\nu F} \cong \nu T_F$*).*

Proof. Since $F[-]$ preserves limits of ω-chains the final coalgebra of F can be constructed as the limit $\varprojlim_{n<\omega} F^n[1]$, and since T preserves this limit the fixed point is also preserved by T, by the dual of theorem 5.6 □

For the construction of least fixed points (or initial algebras) two more preliminary results are needed. First we need to show that the construction of the fixed point can be restricted to $\widehat{\mathscr{G}}$, so that we know that it will be preserved by T.

Proposition 6.5. *The functor $-[-]:\mathscr{G}_{I+1}\times\mathscr{G}_I\to\mathscr{G}_I$ restricts to a functor on the category of cartesian container morphisms, $-[-]:\widehat{\mathscr{G}}_{I+1}\times\widehat{\mathscr{G}}_I\to\widehat{\mathscr{G}}_I$.*

Proof. It is sufficient to show that when $\alpha:F\to F'$ and $\beta:G\to G'$ are both cartesian then so is $\alpha[\beta]$, and indeed it is sufficient to show that $T_\alpha[T_\beta]$ is a cartesian natural transformation. This follows immediately from the fact that T_F preserves pullbacks and that T_α and T_β are cartesian natural transformations. □

Secondly we need to show that $F[-]$ has rank. Assume from now to the end of this section that $\mathbb{C}$ is a finitely[4] accessible category.

Proposition 6.6. *When $\mathbb{C}$ is finitely accessible, every container functor has rank.*

Proof. Let $(A\triangleright B)\in\mathscr{G}_I$ be a container. We first need to establish the result

$$\prod_{i\in I}\left(B_i\Rightarrow\bigvee_{j\in\mathbb{J}}X_{j,i}\right)\cong\bigvee_{j\in\mathbb{J}}\prod_{i\in I}(B_i\Rightarrow X_{j,i})$$

for sufficiently large $\aleph$ and $\aleph$-filtered $\mathbb{J}$, which we do by appealing to two results of Adámek and Rosický (1994). First, we know (from their theorem 2.39) that each functor category $\mathbb{C}^I$ is accessible, and secondly we know from their proposition 2.23 that each functor with an adjoint between accessible categories has rank.

Now since Σ_A preserves colimits we can conclude that $T_{A\triangleright B}$ has rank. □

Corollary 6.7. *For each $F\in\mathscr{G}_{I+1}$ the endofunctor $F[-]$ on $\mathscr{G}_I$ restricts to an endofunctor on $\widehat{\mathscr{G}_I}$ with rank.*

Proof. Let $\aleph$ be the rank of T_F and let D be an $\aleph$-filtered diagram in $\widehat{\mathscr{G}}$. We know that $T_F[-]$ will preserve $\bigvee D$ so we can now repeat the calculation of corollary 6.3 to conclude that $F[-]$ also has rank $\aleph$. □

That containers have least fixed points now follows from corollary 6.7 and theorem 5.7.

Theorem 6.8. *Each $F\in\mathscr{G}_{I+1}$ has a least fixed point $\mu F\in\mathscr{G}_I$ satisfying $T_{\mu F}\cong\mu T_F$.* □

7 Strictly Positive Types

We now return to the point that all strictly positive types can be described as containers.

Definition 7.1. *A* strictly positive type in n variables *(Abel and Altenkirch, 2000) is a type expression (with type variables $X_1,\ldots,X_n$) built up according to the following rules:*

[4] The qualification *finitely* is not strictly necessary here.

- *if K is a constant type (with no type variables) then K is a strictly positive type;*
- *each type variable X_i is a strictly positive type;*
- *if U, V are strictly positive types then so are $U+V$ and $U \times V$;*
- *if K is a constant type and U a strictly positive type then $K \Rightarrow U$ is a strictly positive type;*
- *if U is a strictly positive type in $n+1$ variables then $\mu X.U$ and $\nu X.U$ are strictly positive types in n variables (for X any type variable).*

Note that the type expression for a strictly positive type U can be interpreted as a functor $U : \mathbb{C}^n \to \mathbb{C}$, and indeed we can see that each strictly positive type corresponds to a container in $\mathscr{G}_n$.

Let strictly positive types U, V be represented by containers $(A \triangleright B)$ and $(C \triangleright D)$ respectively, then the table below shows the correspondence between strictly positive types and containers[5].

$$\begin{array}{rr} K \mapsto (K \triangleright 0) & X_i \mapsto (1 \triangleright (\delta_{i,j})_{j \in I}) \\ U + V \mapsto (A + C \triangleright B \mathbin{\hat{+}} D) & U \times V \mapsto (a{:}A, c{:}C \triangleright B(a) \times D(c)) \\ K \Rightarrow U \mapsto (f{:}K \Rightarrow A \triangleright \Sigma k{:}K.B(fk)) & \end{array}$$

The construction of fixed points is a bit more difficult to describe in type-theoretic terms. Let W be represented by $(A \triangleright B, E) \in \mathscr{G}_I$ (see section 6), then for any fixed point C of $T_{A \triangleright E}$ with $\Phi : T_{A \triangleright E} C \cong C$ we can define $C \vdash D_C$ as the initial solution of

$$D_C(\Phi(a,f)) \cong B(a) + \Sigma e{:}E.D_C(fe) \ ; \qquad (*)$$

we can now define

$$\begin{array}{l} \mu X.W \mapsto (\mu X.T_{A \triangleright E} X \triangleright D_{\mu X.T_{A \triangleright E} X}) \\ \nu X.W \mapsto (\nu X.T_{A \triangleright E} X \triangleright D_{\nu X.T_{A \triangleright E} X}) \ . \end{array}$$

All the initial and terminal (co)algebras used above can be constructed explicitly using the results of section 6. It is interesting to note that μ and ν only differ in the type of shapes but that the type of positions can be defined uniformly.

Indeed, consider $F(X) = \mu Y.1 + X \times Y$, then $\mu X.F(X)$ is the type of lists and as we have already observed the type of shapes is isomorphic to $\mathbb{N} \cong \mu X.1 + X$ and the family of positions over n can be conveniently described by $P(n) = \{i \mid i < n\}$. Dually, $\nu X.F(X)$ is the type of lazy (i.e. potentially infinite) lists. The type of shapes is given by $\mathbb{N}^{\mathrm{co}} = \nu X.1 + X$, the *conatural numbers*, which contain a fixed point of the successor $\omega = \mathrm{s}(\omega) : \mathbb{N}^{\mathrm{co}}$. Hence $P(\omega) \cong \mathbb{N}$ and this represents the infinite lists whose elements can be indexed by the natural numbers. Had we used the *terminal* solution of (*) to construct the type of positions, then the representation of infinite lists would incorrectly have an additional *infinite* position.

In the reverse direction it seems that there are containers which do not correspond to strictly positive types. A probable counterexample is the type of nests, defined as the least solution to the equation

$$N(Y) \cong 1 + Y \times N(Y \times Y) \ .$$

[5] We write $\delta_{i,j} \equiv 1$ iff $i = j$ and $\delta_{i,j} \equiv 0$ otherwise.

The datatype N *is* a container since it can be written as $N(X) \cong \Sigma n : \mathbb{N}. X^{2^n-1}$, but it should be possible to show that it is not strictly positive following the argument used in Moggi et al. (1999) to show that the type of square matrices is not regular.

8 Relationships with Shapely Types

In Jay and Cockett (1994) and Jay (1995) "shapely types" (in one parameter) in a category $\mathbb{C}$ are defined to be strong pullback preserving functors $\mathbb{C} \to \mathbb{C}$ equipped with a strong cartesian natural transformation to the list type, where the *list type* is the initial algebra $\mu Y. 1 + X \times Y$.

To see the relationship with containers, note that proposition 2.6.11 of Jacobs (1999) tells us that strong pullback preserving functors are in bijection with fibred pullback preserving functors, and similarly strong natural transformations between such functors correspond to fibred natural transformations. The next proposition will allow us to immediately observe that shapely types are containers.

Proposition 8.1. *Any functor $G \in [\mathbb{C}^I, \mathbb{C}]$ equipped with a cartesian natural transformation $\alpha : G \to T_F$ to a container functor is itself isomorphic to a container functor.*

Proof. Let $F \equiv (A \triangleright B)$ then $(\alpha_1, \mathrm{id}_{\alpha_1^* B}) : (G1 \triangleright \alpha_1^* B) \to (A \triangleright B)$ is a cartesian map in $\mathscr{G}_I$; this yields a cartesian natural transformation $T_{G1 \triangleright \alpha_1^* B} \to T_{A \triangleright B}$. It now follows from the observation that each α_X makes GX the pullback along α_1 of the map $T_{A \triangleright B} X \to A$ that $G \cong T_{G1 \triangleright \alpha_1^* B}$ as required. □

Since the "list type" is given by the container $(n : \mathbb{N} \triangleright [n])$, it immediately follows (when $\mathbb{C}$ is locally cartesian closed) that every shapely type is a container functor.

In the opposite direction, containers which are locally isomorphic to finite cardinals give rise to shapely types. To see this, we follow Johnstone (1977) and refer to the object $[-] \in \mathbb{C}/\mathbb{N}$, which can be constructed as the morphism $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mapping $(n, m) \mapsto n + m + 1$, as the object of *finite cardinals* in $\mathbb{C}$.

Definition 8.2. *An object $A \vdash B$ is* discretely finite *iff there exists a morphism $u : A \to \mathbb{N}$ such that $B \cong u^*[-]$, i.e. each fibre $a : A \vdash B(a)$ is isomorphic to a finite cardinal.*

Say that a container $(A \triangleright B) \in \mathscr{G}_I$ is discretely finite *iff each component B_i for $i \in I$ is discretely finite.*

Note that "discretely finite" is strictly stronger than finitely presentable and other possible notions of finiteness. An immediate consequence of this definition is that the object of finite cardinals is a *generic object* for the category of discretely finite containers, and the following theorem relating shapely types and containers now follows as a corollary.

Theorem 8.3. *In a locally cartesian closed category with a natural number object the category of shapely functors and strong natural transformations is equivalent to the category of discretely finite containers.* □

However, this paper tells us more about shapely types. In particular, containers show how to extend shapely types to cover coinductive types. Finally, the representation result for containers clearly translates into a representation result classifying the polymorphic functions between shapely types.

It interesting to note that the "traversals" of Moggi et al. (1999) do not carry over to containers in general, for example the type $\mathbb{N} \Rightarrow X$ does not effectively traverse over the lifting monad $X \mapsto X + 1$.

References

M. Abbott, T. Altenkirch, N. Ghani, and C. McBride. Derivatives of containers. URL `http://www.cs.nott.ac.uk/~{}txa/`. Submitted for publication, 2003.

A. Abel and T. Altenkirch. A predicative strong normalisation proof for a λ-calculus with interleaving inductive types. In *Types for Proof and Programs,TYPES '99*, volume 1956 of *Lecture Notes in Computer Science*, 2000.

J. Adámek and V. Koubek. Least fixed point of a functor. *Journal of Computer and System Sciences*, 19:163–178, 1979.

J. Adámek and J. Rosický. *Locally Presentable and Accessible Categories*. Number 189 in London Mathematical Society Lecture Note Series. Cambridge University Press, 1994.

T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *IFIP Working Conference on Generic Programming*, 2002.

F. Borceux. *Handbook of Categorical Algebra 2*. Cambridge University Press, 1994.

A. Carboni and P. Johnstone. Connected limits, familial representability and Artin glueing. *Math. Struct. in Comp. Science*, 5:441–459, 1995.

R. Hasegawa. Two applications of analytic functors. *Theoretical Computer Science*, 272(1-2): 112–175, 2002.

M. Hofmann. Syntax and semantics of dependent types. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, volume 14, pages 79–130. Cambridge University Press, Cambridge, 1997.

P. Hoogendijk and O. de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, 2000.

B. Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. Elsevier, 1999.

C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

C. B. Jay and J. R. B. Cockett. Shapely types and shape polymorphism. In *ESOP '94: 5th European Symposium on Programming*, Lecture Notes in Computer Science, pages 302–316. Springer-Verlag, 1994.

P. T. Johnstone. *Topos Theory*. Academic Press, 1977.

C. McBride. The derivative of a regular type is its type of one-hole contexts. URL `http://www.dur.ac.uk/c.t.mcbride/`. 2001.

E. Moggi, G. Bellè, and C. B. Jay. Monads, shapely functors and traversals. *Electronic Notes in Theoretical Computer Science*, 29, 1999.

P. Taylor. *Practical Foundations of Mathematics*. Cambridge University Press, 1999.

Verification of Probabilistic Systems with Faulty Communication

Parosh Aziz Abdulla[1] and Alexander Rabinovich[2]

[1] Uppsala University, Sweden
[2] Tel Aviv University, Israel

Abstract. Many protocols are designed to operate correctly even in the case where the underlying communication medium is faulty. To capture the behaviour of such protocols, *lossy channel systems (LCS)* [AJ96b] have been proposed. In an LCS the communication channels are modelled as FIFO buffers which are unbounded, but also unreliable in the sense that they can nondeterministically lose messages.
Recently, several attempts [BE99, ABIJ00] have been made to study *probabilistic Lossy Channel Systems (PLCS)* in which the probability of losing messages is taken into account. In this paper, we consider a variant of PLCS which is more realistic than those studied in [BE99, ABIJ00]. More precisely, we assume that during each step in the execution of the system, each message may be lost with a certain predefined probability. We show that for such systems the following model checking problem is decidable: to verify whether a given property definable by finite state ω-automata holds with probability one. We also consider other types of faulty behavior, such as corruption and duplication of messages, and insertion of new messages, and show that the decidability results extend to these models.

1 Introduction

Finite state machines which communicate through unbounded buffers have been popular in the modelling of communication protocols [BZ83, Boc78]. One disadvantage with such a model is that it has the full computation power of Turing machines [BZ83], implying undecidability of all nontrivial verification problems. On the other hand, many protocols are designed to operate correctly even in the case where the underlying communication medium is faulty. To capture the behaviour of such protocols, *lossy channel systems (LCS)* [AJ96b] have been proposed as an alternative model. In an LCS the communication channels are modelled as FIFO buffers which are unbounded but also unreliable in the sense that they can nondeterministically lose messages. For LCS it has been shown that the reachability problem is decidable [AJ96b] while progress properties are undecidable [AJ96a].

Since we are dealing with unreliable communication media, it is natural to deal with models where the probability of losing messages is taken into account.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 39–53, 2003.

Recently, several attempts [BE99, ABIJ00] have been made to study *probabilistic Lossy Channel Systems (PLCS)* which introduce randomization into the behaviour of LCS. The decidability of model checking for the proposed models depend heavily on the semantics provided. The works in [BE99, ABIJ00] define different semantics for PLCS depending on the manner in which the messages may be lost inside the channels.

Baier and Engelen [BE99] consider a model where it is assumed that at most single message may be lost during each step of the execution of the system. They show decidability of model checking under the assumption that the probability of losing messages is at least 0.5. This implies that, along each computation of the system, there are infinitely many points where the channels of the system are empty, and therefore the model checking problem reduces to checking decidable properties of the underlying (non-probabilistic) LCS.

The model in [ABIJ00] assumes that messages can only be lost during send operations. Once a message is successfully sent to a channel, it continues to reside inside the channel until it is removed by a receive operation. Both the reachability and repeated reachability problems are shown to be undecidable for this model of PLCS. The idea of the proof is to choose sufficiently low probabilities for message losses to enable the system to simulate the behaviour of (non-probabilistic) systems with perfect channels.

In this paper, we consider a variant of PLCS which are more realistic than that in [BE99, ABIJ00]. More precisely, we assume that, during each step in the execution of the system, each message may be lost with a certain predefined probability. This means that the probability of losing a certain message will not decrease with the length of the channels (as it is the case with [BE99]). Thus, in contrast to [BE99] our method is not dependent on the precise transition probabilities for establishing the qualitative properties of the system. For this model, we show decidability of both the reachability and repeated reachability problems.

The decidability results are achieved in two steps. First, we prove general theorems about (infinite state) Markov chains which serve as sufficient conditions for decidability of model checking. To do that, we introduce the concept of *attractor sets*: all computations of the system are guaranteed to eventually reach the attractor. The existence of finite attractors imply that deciding reachability and repeated reachability in the PLCS can be reduced to checking reachability problems in the underlying LCS. Next, we show that all PLCS, when interpreted according to our semantics, have finite attractors. More precisely, we prove the existence of an attractor defined by the set of all configuration where the sizes of channels are bound by some natural number. This natural number can be derived from the predefined probability given to the loss of messages. In fact, for the systems considered in [BE99] this bound is equal to 0, and therefore the decidability results in [BE99] can be seen as a consequence of the properties we show for attractors.

We also show that our decidability results extend to PLCS with different sources of unreliability [CFI96], such as duplication, corruption, and insertion

combined with lossiness Furthermore, we show how to extend our decidability results to more general properties specified by finite state automata or equivalently by formulas of Monadic Logic of Order.

Remark Bertrand and Schnoebelen [BS03] have independently obtained what essentially amounts to Theorem 1 in this paper.

Outline In the next two Sections we give basics of transition systems and Markov chains respectively. In Section 4 we present sufficient conditions for checking reachability and repeated reachability for Markov chains. In Section 5 we extract from these conditions algorithms for PLCS. In Section 6 we consider models involving different sources of unreliability combined with lossiness. In Section 7 we generalize our results to verification of the properties definable by ω-behavior of finite state automata (or equivalently formulas in the Monadic Logic of Order). Finally, we give conclusions and directions for future work in Section 8.

2 Transition Systems

In this section, we recall some basic concepts of transition systems.

A *transition system* T is a pair $(S, \longrightarrow)$ where S is a (potentially) infinite set of states, and $\longrightarrow$ is a binary relation on S. We write $s_1 \longrightarrow s_2$ to denote that $(s_1, s_2) \in \longrightarrow$ and use $\stackrel{*}{\longrightarrow}$ to denote the reflexive transitive closure of $\longrightarrow$. We say that s_2 is *reachable* from s_1 if $s_1 \stackrel{*}{\longrightarrow} s_2$. For sets $Q_1, Q_2 \subseteq S$, we say that Q_2 is *reachable* from Q_1, denoted $Q_1 \stackrel{*}{\longrightarrow} Q_2$, if there are $s_1 \in Q_1$ and $s_2 \in Q_2$ with $s_1 \stackrel{*}{\longrightarrow} s_2$. A *path* p from s to s' is of the form $s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow s_n$, where $s_0 = s$ and $s_n = s'$. We say that p is *simple* if there are no i, j with $i \neq j$ and $s_i = s_j$. For a set $Q \subseteq S$, we say that p *reaches* Q if $s_i \in Q$ for some $i : 0 \leq i \leq n$. For $Q_1, Q_2 \subseteq S$, we define the set $\mathit{Until}(Q_1, Q_2)$ to the set of all states s such that there is a path $s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow s_n$ from s satisfying the following property: there is an $i : 0 \leq i \leq n$ such that $s_i \in Q_2$ and for each $j : 0 \leq j < i$ we have $s_j \in Q_1$.

For $Q \subseteq S$, we define the *graph* of Q, denoted $\mathit{Graph}(Q)$, to be the transition system $\left(Q, \longrightarrow'\right)$ where $s_1 \longrightarrow' s_2$ iff $s_1 \stackrel{*}{\longrightarrow} s_2$.

A *strongly connected component (SCC)* in T is a maximal set $C \subseteq S$ such that $s_1 \stackrel{*}{\longrightarrow} s_2$ for each $s_1, s_2 \in C$. We say that C is a *bottom SCC (BSCC)* if there is no other SCC C_1 in T with $C \stackrel{*}{\longrightarrow} C_1$. In other words, the BSCCs are the leafs in the acyclic graph of SCCs (ordered by reachability).

We shall later refer to the following two problems for transition systems

Reachability
Instance A transition system $T = (S, \longrightarrow)$, and sets $Q_1, Q_2 \subseteq S$.
Question Is Q_2 reachable from Q_1?

Until
Instance A transition system $T = (S, \longrightarrow)$, a state s, and sets $Q_1, Q_2 \subseteq S$.
Question Is $s \in \mathit{Until}(Q_1, Q_2)$?

3 Markov Chains

In this section, we introduce (potentially infinite state) *Markov chains*.

A *Markov chain* M is a pair (S, P) where S is a (potentially infinite) set of states and P is a mapping from $S \times S$ to the set $[0, 1]$, such that $\sum_{s' \in S} P(s, s') = 1$, for each $s \in S$. A *computation* π *(from* s_0*)* of M is an infinite sequence $s_0, s_1, \ldots$ of states. We use $\pi(i)$ to denote s_i.

A Markov chain induces a transition system, where the transition relation consists of pairs of states related by positive probabilities. Formally, the *underlying transition system* of M is $(S, \longrightarrow)$ where $s_1 \longrightarrow s_2$ iff $P(s_1, s_2) > 0$. In this manner, the concepts defined for transition systems can be lifted to Markov chains. For instance, an SCC in M is a SCC in the underlying transition system.

A Markov chain (S, P) induces a natural measure on the set of computations from every state s.

Let us recall some basic notions from probability theory.

A *measurable space* is a pair (Ω, Δ) consisting of a non empty set Ω and a σ-algebra Δ of its subsets that are called *measurable sets* and represent random events in probability context. A *σ-algebra* over Ω contains Ω and is closed under complementation and countable union. Adding to a measurable space a *probability measure* $Prob : \Delta \rightarrow [0, 1]$ such that $Prob(\Omega) = 1$ and that is countably additive, we get a *probability space* $(\Omega, \Delta, Prob)$.

Consider a state s of a Markov chain (S, P). On the sets of computations that start at s, the probabilistic space is defined as follows:

Probabilistic space $(\Omega, \Delta, Prob)$(see [KSK66]) : $\Omega = sS^\omega$ is the set of all infinite sequences of states starting from s, Δ is the σ-algebra generated by the basic cylindric sets $D_u = uS^\omega$, for every $u \in sS^*$, and the probability measure $Prob$ is defined by $Prob(D_u) = \prod_{i=0,\ldots,n-1} P(s_i, s_{i+1})$ where $u = s_0 s_1 ... s_n$; it is well-known that this measure is extended in a unique way to the elements of the σ-algebra generated by the basic cylindric sets.

4 Reachability Analysis for Markov Chains

In this section we explain how to check reachability and repeated reachability for Markov chains. We show how to reduce qualitative properties of the above two types into the analysis of the underlying (non-probabilistic) transition system of the Markov chain.

In the rest of this section, we assume a Markov chain $M = (S, P)$ with an underlying transition system $T = (S, \longrightarrow)$.

Consider a set $Q \subseteq S$ of states and a computation π. We say that π *reaches* Q if there is an $i \geq 0$ with $\pi(i) \in Q$. We say that π *repeatedly reaches* Q if there are infinitely many i with $\pi(i) \in Q$. Let s be a state in S. We define the probability of Q being (repeatedly) reachable from s by

$Prob\,\{\pi |\ \pi$ is a computation from s and π (repeatedly) reaches $Q\}$.

We consider the following two problems for Markov chains:

Probabilistic Reachability
Instance A Markov chain $M = (S, P)$, a state $s \in S$, and a set $Q \subseteq S$.
Question Is Q reachable from s with probability one?

Probabilistic Repeated Reachability
Instance A Markov chain $M = (S, P)$, a state $s \in S$, and a set $Q \subseteq S$.
Question Is Q repeatedly reachable from s with probability one?

In the above problems, we do not assume that Markov chains are finite. Hence these are not instances of algorithmic problems. In Sections 5-7 we consider reachability and repeated reachability problems when countable Markov chains are described by probabilistic lossy channel systems. For such finite descriptions we investigate the corresponding algorithmic problems.

We introduce a central concept which we use in our solution for the probabilistic (repeated) reachability problem, namely that of *attractors*.
Definition [attractors] A set $A \subseteq S$ is said to be an *attractor*, if for each $s \in S$, the set A is reachable from s with probability one.

In other words, regardless of the state in which we start, we are guaranteed that we will eventually enter the attractor.

We consider two preliminary lemmas which are derived from the standard properties of recurrent classes. The Lemma below describes a property of BSCCs of the graph of a finite attractor A, which will make use of in our algorithms (to prove Lemma 2 and Lemma 3).

Lemma 1. *Consider a finite attractor A, a BSCC C in Graph(A), and a state $s \in C$. Then, for every $s' \in C$, the probability that s' is repeatedly reachable from s is one.*

The following Lemma enables us to construct an algorithm for solving the probabilistic reachability problem.

Lemma 2. *Consider a finite attractor A, a state $s \in S$, and a set $Q \subseteq S$. Then, Q is reachable from s with probability one iff for each BSCC C in Graph(A), if C is reachable from s then either*

- *Q is reachable from C; or*
- *For every finite simple path in T from s, if p reaches C then p also reaches Q.*

From Lemma 2 we conclude that we can define a scheme for solving the reachability problem as follows.

Scheme – Probabilistic Reachability

Input Markov chain $M = (S, P)$ with an underlying transition system $T = (S, \longrightarrow)$, a state $s \in S$, and a set $Q \subseteq S$.
Output Is Q reachable from s with probability one?
begin
1. construct a finite attractor A
2. construct $Graph(A)$
3. **for** each BSCC C in $Graph(A)$ which is reachable from s
   3a. **if** $\neg \left(C \xrightarrow{*} Q \right)$ **and** $s \in Until(\neg Q, C)$ **then** return(false)
4. return(true)

end

The following Lemma enables us to construct an algorithm for solving the probabilistic repeated reachability problem.

Lemma 3. *Consider a finite attractor A, a state $s \in S$, and a set $Q \subseteq S$. Then, Q is repeatedly reachable form s with probability one iff the reachability of C from s implies the reachability of Q from C, for each BSCC C in $Graph(A)$.*

From Lemma 3 we conclude that we can define a scheme for solving the repeated reachability problem as follows.

3a. **if** $\neg \left(C \xrightarrow{*} Q \right)$ **then** return(false)

The correctness of the two schemes follows immediately from Lemma 2 and Lemma 3. Furthermore, we observe that, in order to obtain algorithms for checking the reachability and repeated reachability problems, we need the following three effectiveness properties for the operations involved:

1. Existence and computability of a finite attractor. This condition is necessary for computing the set A.
2. Decidability of the reachability problem for the underlying class of transition systems T. This condition is necessary for computing $Graph(A)$ and for checking the relation $C \xrightarrow{*} Q$.
3. Decidability of the until problem for the underlying class of transition systems. This condition is only needed in the reachability algorithm.

5 Lossy Channel Systems

In this section we consider (probabilistic) lossy channel systems: processes with a finite set of local states operating on a number of unbounded and unreliable channels. We use the scheme defined in Section 4 to solve the problem of whether a set of local states is (repeatedly) reachable from a given initial state with probability one.

Lossy Channel Systems A *lossy channel system* consists of a finite state process operating on a finite set of channels each of which behaves as a FIFO buffer which is unbounded and unreliable in the sense that it can nondeterministically lose messages. Formally, a *lossy channel system (LCS)* $\mathcal{L}$ is a tuple $(\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T})$ where $\mathtt{S}$ is a finite set of *local states*, $\mathtt{C}$ is a finite set of *channels*, $\mathtt{M}$ is a finite *message alphabet*, and $\mathtt{T}$ is a set of *transitions* each of the form $(\mathtt{s}_1, \mathtt{op}, \mathtt{s}_2)$, where $\mathtt{s}_1, \mathtt{s}_2 \in \mathtt{S}$, and $\mathtt{op}$ is an *operation* of one of the forms $\mathtt{c!m}$ (sending message $\mathtt{m}$ to channel $\mathtt{c}$), or $\mathtt{c?m}$ (receiving message $\mathtt{m}$ from channel $\mathtt{c}$). A *global state* s is of the form $(\mathtt{s}, \mathtt{w})$ where $\mathtt{s} \in \mathtt{S}$ and $\mathtt{w}$ is a mapping from $\mathtt{C}$ to $\mathtt{M}^*$.

For words $x, y \in \mathtt{M}^*$, we use $x \bullet y$ to denote the concatenation of x and y. We write $x \preceq y$ to denote that x is a (not necessarily contiguous) substring of y. By Higman's Lemma [Hig52] it follows that $\preceq$ is a well quasi-ordering, i.e., for each infinite sequence $x_0, x_1, x_2, \ldots$ there are i and j with $i < j$ and $x_i \preceq x_j$. We use $|x|$ to denote the length of x, and use $x(i)$ to denote the i^{th} element of x where $i : 1 \leq i \leq |x|$. For $\mathtt{w}_1, \mathtt{w}_2 \in (\mathtt{C} \mapsto \mathtt{M}^*)$, we use $\mathtt{w}_1 \preceq \mathtt{w}_2$ to denote that $\mathtt{w}_1(\mathtt{c}) \preceq \mathtt{w}_2(\mathtt{c})$ for each $\mathtt{c} \in \mathtt{C}$, and define $|\mathtt{w}| = \sum_{\mathtt{c} \in \mathtt{C}} |\mathtt{w}(\mathtt{c})|$. We also extend $\preceq$ to a relation on $\mathtt{S} \times (\mathtt{C} \mapsto \mathtt{M}^*)$, where $(\mathtt{s}_1, \mathtt{w}_1) \preceq (\mathtt{s}_2, \mathtt{w}_2)$ iff $\mathtt{s}_1 = \mathtt{s}_2$ and $\mathtt{w}_1 \preceq \mathtt{w}_2$.

The LCS $\mathcal{L}$ induces a transition system $(S, \longrightarrow)$, where S is the set of global states, i.e., $S = (\mathtt{S} \times (\mathtt{C} \mapsto \mathtt{M}^*))$, and $(\mathtt{s}_1, \mathtt{w}_1) \longrightarrow (\mathtt{s}_2, \mathtt{w}_2)$ iff one of the following conditions is satisfied

- There is a $\mathtt{t} \in \mathtt{T}$, where $\mathtt{t}$ is of the form $(\mathtt{s}_1, \mathtt{c!m}, \mathtt{s}_2)$ and $\mathtt{w}_2$ is the result of appending $\mathtt{m}$ to the end of $\mathtt{w}_1(\mathtt{c})$.
- There is a $\mathtt{t} \in \mathtt{T}$, where $\mathtt{t}$ is of the form $(\mathtt{s}_1, \mathtt{c?m}, \mathtt{s}_2)$ and $\mathtt{w}_1$ is the result of removing $\mathtt{m}$ from the head of $\mathtt{w}_2(\mathtt{c})$.
- Furthermore, if $(\mathtt{s}_1, \mathtt{w}_1) \longrightarrow (\mathtt{s}_2, \mathtt{w}_2)$ according to one of the previous two rules then $(\mathtt{s}_1, \mathtt{w}_1) \longrightarrow (\mathtt{s}'_2, \mathtt{w}'_2)$ for each $(\mathtt{s}'_2, \mathtt{w}'_2) \preceq (\mathtt{s}_2, \mathtt{w}_2)$.

In the first two cases we define $\mathtt{t}(\mathtt{s}_1, \mathtt{w}_1) = (\mathtt{s}_2, \mathtt{w}_2)$.

A transition $(\mathtt{s}_1, \mathtt{op}, \mathtt{s}_2)$ is said to be *enabled* at $(\mathtt{s}, \mathtt{w})$ if $s = s_1$ and either

- $\mathtt{op}$ is of the form $\mathtt{c!m}$; or
- $\mathtt{op}$ is of the form $\mathtt{c?m}$ and $\mathtt{w}(\mathtt{c}) = \mathtt{m} \bullet x$, for some $x \in \mathtt{M}^*$.

We defined $\mathit{enabled}(\mathtt{s}, \mathtt{w}) = \{\mathtt{t} |\ \mathtt{t} \text{ is enabled at } (\mathtt{s}, \mathtt{w})\}$. In the sequel, we assume that for all $(\mathtt{s}, \mathtt{w})$, the set $\mathit{enabled}(\mathtt{s}, \mathtt{w})$ is not empty. This is guaranteed for instance, by requiring that for any local state $\mathtt{s}_1$ there are $\mathtt{c}$, $\mathtt{m}$, and $\mathtt{s}_2$ with $(\mathtt{s}_1, \mathtt{c!m}, \mathtt{s}_2) \in \mathtt{T}$

Remark on notation We use $\mathtt{s}$ and $\mathtt{S}$ to range over local states and sets of local states respectively. On the other hand, we s and S to range over states and sets of states of the induced transition system (states of the transition system are global states of the LCS)

For the rest of this section we assume an LCS $(\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T})$.

For $\mathtt{Q} \subseteq \mathtt{S}$, we define a $\mathtt{Q}$-*state* to be a state of the form $(\mathtt{s}, \mathtt{w})$ where $\mathtt{s} \in \mathtt{Q}$. A set $Q \subseteq S$ is said to be *upward closed* if $s_1 \in Q$ and $s_1 \preceq s_2$ imply $s_2 \in Q$. Notice that, for any $\mathtt{Q} \subseteq \mathtt{S}$, the set of $\mathtt{Q}$-states is an upward closed set.

In [AJ96b], algorithms are given which shows the following decidability results for LCS:

Lemma 4. *For states s_1 and s_2, it is decidable whether s_2 is reachable from s_1.*

Lemma 5. *For a state s and a set $\mathtt{Q} \subseteq \mathtt{S}$, it is decidable whether the set of $\mathtt{Q}$-states is reachable from s.*

Decidability of the corresponding until problem follows from a straightforward modification of the reachability algorithm of [AJ96b]. This gives the following.

Lemma 6. *For a state s, a set $\mathtt{Q}_1 \subseteq \mathtt{S}$, and a finite set Q_2 of states, it is decidable whether $s \in Until(\neg Q_1, Q_2)$, where Q_1 is the set of $\mathtt{Q}_1$-states.*

Probabilistic Lossy Channel Systems A *probabilistic lossy channel system (PLCS)* $\mathcal{L}$ is of the form $(\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T}, \lambda, w)$, where $(\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T})$ is an LCS, $\lambda \in [0, 1]$, and w is a mapping from $\mathtt{T}$ to the natural numbers. Intuitively, we derive a Markov chain from the PLCS $\mathcal{L}$ by assigning probabilities to the transitions of the underlying transition system $(\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T})$. The probability of performing a transition $\mathtt{t}$ from a global state $(\mathtt{s}, \mathtt{w})$ is determined by the weight $w(\mathtt{t})$ of $\mathtt{t}$ compared to the weights of the other transitions which are enabled at $(\mathtt{s}, \mathtt{w})$. Furthermore, after performing each transition, each message which resides inside one of the channels may be lost with a probability λ. This means that the probability of reaching $(\mathtt{s}_2, \mathtt{w}_2)$ from $(\mathtt{s}_1, \mathtt{w}_1)$ is equal to (the sum over all $(s_3, \mathtt{w}_3)$ of) the probability of reaching some $(s_3, \mathtt{w}_3)$ from $(\mathtt{s}_1, \mathtt{w}_1)$ through performing a transition of the underlying LCS, multiplied by the probability of reaching $(\mathtt{s}_2, \mathtt{w}_2)$ from $(\mathtt{s}_3, \mathtt{w}_3)$ through the loss of messages. Now, we show how to derive these probabilities from the definition of $\mathcal{L}$.

First, we compute probabilities of reaching states through the loss of messages. For $x, y \in \mathtt{M}^*$, we define $\#(x, y)$ to be the size of the set

$$\{(i_1, \ldots, i_n) \mid i_1 < \cdots < i_n \text{ and } x = y(i_1) \bullet \cdots \bullet y(i_n)\}$$

In other words, $\#(x, y)$ is the number of the different ways in which we can delete symbols in the word y in order to obtain x. We also define $P_L(x, y) = \#(x, y) \cdot \lambda^{|y|-|x|} \cdot (1 - \lambda)^{|x|}$. For $\mathtt{w}_1, \mathtt{w}_2 \in (\mathtt{C} \mapsto \mathtt{M}^*)$, we define $P_L(\mathtt{w}_1, \mathtt{w}_2) = \prod_{\mathtt{c} \in \mathtt{C}} P_L(\mathtt{w}_1(\mathtt{c}), \mathtt{w}_2(\mathtt{c}))$. Intuitively, $P_L(\mathtt{w}_1, \mathtt{w}_2)$ defines the probability by which $\mathtt{w}_2$ can change to $\mathtt{w}_1$ through loss of messages during a single step of the execution of the system. Notice that $P_L(\mathtt{w}_1, \mathtt{w}_2) = 0$ in case $\mathtt{w}_1 \not\preceq \mathtt{w}_2$. We take $P_L((\mathtt{s}_1, \mathtt{w}_1), (\mathtt{s}_2, \mathtt{w}_2)) = P_L(\mathtt{w}_1, \mathtt{w}_2)$ if $\mathtt{s}_1 = \mathtt{s}_2$, and $P_L((\mathtt{s}_1, \mathtt{w}_1), (\mathtt{s}_2, \mathtt{w}_2)) = 0$ otherwise. We define $w(\mathtt{s}, \mathtt{w}) = \sum_{\mathtt{t} \in enabled(\mathtt{s}, \mathtt{w})} w(\mathtt{t})$.

The PLCS $\mathcal{L}$ induces a Markov chain (S, P), where $S = (\mathtt{S} \times (\mathtt{C} \mapsto \mathtt{M}^*))$ and $P((\mathtt{s}_1, \mathtt{w}_1), (\mathtt{s}_2, \mathtt{w}_2)) = \sum_{\mathtt{t} \in \mathtt{T}} ((w(\mathtt{t})/w(\mathtt{s}_1, \mathtt{w}_1)) \cdot P_L(\mathtt{t}(\mathtt{s}_1, \mathtt{w}_1), (\mathtt{s}_2, \mathtt{w}_2)))$. Notice that this is well-defined by the assumption that there are no deadlock states.

We instantiate the reachability problems considered in Section 3 and Section 4 to PLCS.

Below, we assume a PLCS $\mathcal{L} = (\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T}, \lambda, w)$ inducing a Markov chain $M = (S, P)$ with an underlying transition system $T = (S, \longrightarrow)$.

We shall consider the probabilistic (repeated) reachability problem for PLCS. We check whether an upward closed set, represented by its minimal elements,

is (repeatedly) reachable from a given initial state with probability one. We show that the (repeated) reachability problem instantiated in this manner fulfills the three conditions required for effective implementation of the probabilistic (repeated) reachability schemes of Section 4.

The following Lemma shows that we can always construct a finite attractor in a PLCS.

Lemma 7. *For each λ, w, and PLCS $(\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T}, \lambda, w)$, thse set $\{(\mathtt{s}, \mathtt{w}) \mid |\mathtt{w}| = 0\}$ is an attractor.*

From Lemma 4, and the fact that the transition system underlying a PLCS $(\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T}, \lambda, w)$ is independent on λ we obtain:

Lemma 8. *For each PLCS $(\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T}, \lambda, w)$, we can compute the graph $Graph(A)$ of a finite set A.*

Furthermore, for two PLCS $\mathcal{L} = (\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T}, \lambda, w)$ and $\mathcal{L}' = (\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T}, \lambda', w')$ which differ only by probabilities, If $\lambda, \lambda' > 0$ and $w(\mathtt{t}) > 0$ iff $w'(\mathtt{t}) > 0$ then A has the same graph in both PLCS. Now we are ready to solve Probabilistic Reachability and Probabilistic Repeated Reachability problems for PLCS.

Probabilistic Reachability for PLCS
Instance An PLCS $M = (\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T}, \lambda, w)$ a state s, and a set $\mathtt{Q} \subseteq \mathtt{S}$.
Question Is the set of $\mathtt{Q}$-states is reachable from s with probability one?

Probabilistic Repeated Reachability
Instance An PLCS $M = (\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T}, \lambda, w)$ a state s, and a set $\mathtt{Q} \subseteq \mathtt{S}$.
Question Is the set of $\mathtt{Q}$-states is repeatedly reachable from s with probability one?

From the results of Section 4 and Lemma 8, Lemma 5, Lemma 7, and Lemma 6 we get the following.

Theorem 1. *Probabilistic Reachability and Probabilistic Repeated Reachability are decidable for PLCS. .*

Remark In our definition of LCS and PLCS, we assume that messages are lost only after performing non-lossy transitions. Our analysis can be modified in a straightforward manner to deal with the case where losses occur before, and the case where losses occur both before and after non-lossy transitions.

6 Duplication, Corruption, and Insertion

We consider PLCS with different sources of unreliability such as duplication, corruption, and insertion combined with lossiness.
Duplication We analyze a variant of PLCS, where we add another source of unreliability; namely a message inside a channel may be duplicated [CFI96].

An LCS $\mathcal{L}$ with *duplication errors* is of the same form $(\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T})$ as an LCS. We define the behaviour of $\mathcal{L}$ as follows. For $a \in \mathtt{M}$, we use a^n to denote the

concatenation of n copies of a. For $x = a_1 a_2 \cdots a_n$ with $x \in \mathtt{M}^*$, we define $\mathit{duplicate}(x)$ to be the set

$\{b_1 b_2 \cdots b_n \mid$ either $b_i = a_i$ or $b_i = a_i^2$ for each $i : 1 \leq i \leq n\}$

In other words, we get each member of $\mathit{duplicate}(x)$ by duplicating some of the elements of x. We extend the definition of *duplicate* to $\mathtt{S} \times (\mathtt{C} \mapsto \mathtt{M}^*)$ in a similar manner to Section 5. The transition relation of an LCS $\mathcal{L}$ with duplication errors is enlargement of that of the corresponding standard LCS in the sense that:

- If $(\mathtt{s}_1, \mathtt{w}_1) \longrightarrow (\mathtt{s}_2, \mathtt{w}_2)$ according to the definition of Section 5 then $(\mathtt{s}_1, \mathtt{w}_1) \longrightarrow (\mathtt{s}'_2, \mathtt{w}'_2)$ for each $(\mathtt{s}'_2, \mathtt{w}'_2) \in \mathit{duplicate}(\mathtt{s}_2, \mathtt{w}_2)$.

In [CFI96], it is shown that the reachability problem is decidable for LCS with duplication errors. The reachability algorithm can be modified in a similar manner to Section 5 to solve the until problem. Hence we have

Lemma 9. *Given LCS with duplication errors.*

1. *For states s_1 and s_2, it is decidable whether s_2 is reachable from s_1 [CFI96]. Hence, Graph(A) is computable for any finite set A of states.*
2. *For a state s and a set $\mathtt{Q} \subseteq \mathtt{S}$, it is decidable whether the set of $\mathtt{Q}$-states is reachable from s [CFI96].*
3. *For a state s, a set $\mathtt{Q}_1 \subseteq \mathtt{S}$, and a finite set Q_2 of states, it is decidable whether $s \in \mathit{Until}(\neg Q_1, Q_2)$, where Q_1 is the set of $\mathtt{Q}_1$-states.*

A PLCS with *duplication errors* is of the form $(\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T}, \lambda, w, \lambda_D)$, where $(\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T}, \lambda, w)$ is a PLCS, and $\lambda_D \in [0, 1]$. The value of λ_D represents the probability by which any given message is duplicated inside the channels.

To obtain the Markov chain induced by a PLCS with duplication errors, we compute probabilities of reaching states through duplication of messages. For $x, y \in \mathtt{M}^*$, where $x = a_1 a_2 \cdots a_n$, we define $\#_D(x, y)$ to be the size of the set $\{(i_1, \ldots, i_n) \mid 1 \leq i_j \leq 2$ and $y = a_1^{i_1} a_2^{i_2} \cdots a_n^{i_n}\}$. In other words, $\#_D(x, y)$ is the number of the different ways in which we can duplicate symbols in the word x in order to obtain y. In a similar manner to the case of losing messages (Section 5), we define $P_D(x, y) = \#_D(x, y) \cdot \lambda_D^{|y|-|x|} \cdot (1-\lambda_D)^{|x|}$, and $P_D(\mathtt{w}_1, \mathtt{w}_2) = \prod_{\mathtt{c} \in \mathtt{C}} P_D(\mathtt{w}_1(\mathtt{c}), \mathtt{w}_2(\mathtt{c}))$. The PLCS $\mathcal{L}$ with duplication errors induces a Markov chain (S, P'_D), where $S = (\mathtt{S} \times (\mathtt{C} \mapsto \mathtt{M}^*))$ and

$P'_D((\mathtt{s}_1, \mathtt{w}_1), (\mathtt{s}_2, \mathtt{w}_2)) = \sum_{(\mathtt{s}_3, \mathtt{w}_3)} P((\mathtt{s}_1, \mathtt{w}_1), (\mathtt{s}_3, \mathtt{w}_3)) \cdot P_D((\mathtt{s}_3, \mathtt{w}_3), (\mathtt{s}_2, \mathtt{w}_2))$

where P has the same definition as in Section 5. Notice that the sum is computable since the set $\{(\mathtt{s}_3, \mathtt{w}_3) \mid P((\mathtt{s}_1, \mathtt{w}_1), (\mathtt{s}_3, \mathtt{w}_3)) \neq 0\}$ is finite and computable.

Lemma 10. *For each λ, w, λ_D, and PLCS $(\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T}, \lambda, w, \lambda_D)$ with $\lambda_D < \lambda$, the set $\{(\mathtt{s}, \mathtt{w}) \mid |\mathtt{w}| = 0\}$ is an attractor.*

Using a similar reasoning to Section 5, we derive from Lemma 9 and Lemma 10

Theorem 2. *Probabilistic Reachability and Probabilistic Repeated Reachability are decidable for PLCS with duplication errors when $\lambda_D < \lambda$.*

Corruption We consider LCS with *corruption errors*, i.e., a message inside a channel may be changed to any other message. We extend the semantics of LCS to include corruption errors in the same manner as we did above for duplication errors. For $x \in \mathtt{M}^*$, we define $\mathit{Corrupt}(x)$ to be the set $\{y|\ |y| = |x|\}$, i.e., we get a member of $\mathit{Corrupt}(x)$ by changing any number of symbols in x to another symbol in $\mathtt{M}$. We extend the definition to $\mathtt{S} \times (\mathtt{C} \mapsto \mathtt{M}^*)$ in the same manner as before. Furthermore, we enlarge the transition transition of an LCS:

- If $(\mathtt{s}_1, \mathtt{w}_1) \longrightarrow (\mathtt{s}_2, \mathtt{w}_2)$ according to the definition of Section 5 then $(\mathtt{s}_1, \mathtt{w}_1) \longrightarrow (\mathtt{s}'_2, \mathtt{w}'_2)$ for each $(\mathtt{s}'_2, \mathtt{w}'_2) \in \mathit{Corrupt}(\mathtt{s}_2, \mathtt{w}_2)$.

Decidability of the reachability problem for LCS with corruption errors follows from the fact $(\mathtt{s}_1, \mathtt{w}_1) \stackrel{*}{\longrightarrow} (\mathtt{s}_2, \mathtt{w}_2)$ implies $(\mathtt{s}_1, \mathtt{w}_1) \stackrel{*}{\longrightarrow} (\mathtt{s}_2, \mathtt{w}_3)$ for each $\mathtt{w}_3$ with $|\mathtt{w}_3(\mathtt{c})| = |\mathtt{w}_2(\mathtt{c})|$ for all $\mathtt{c} \in \mathtt{C}$. This implies that the only relevant information to consider about the channels in the reachability algorithm is the length of the channels. In other words, the problem is reduced to ¡a special case of LCS systems where the set $\mathtt{M}$ can be considered to be a singleton. The until problem can be solved in a similar manner. Hence,

Lemma 11. *Given LCS with corruption errors.*

1. *For states s_1 and s_2, it is decidable whether s_2 is reachable from s_1. Hence, $\mathit{Graph}(A)$ is computable for any finite set A of states.*
2. *For a state s and a set $\mathtt{Q} \subseteq \mathtt{S}$, it is decidable whether the set of $\mathtt{Q}$-states is reachable from s.*
3. *For a state s, a set $\mathtt{Q}_1 \subseteq \mathtt{S}$, and a finite set Q_2 of states, it is decidable whether $s \in \mathit{Until}(\neg Q_1, Q_2)$, where Q_1 is the set of $\mathtt{Q}_1$-states.*

A PLCS with *corruption errors* is of the form $(\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T}, \lambda, w, \lambda_C)$, where $\lambda_D \in [0, 1]$ represents the probability by which any given message is corrupted to some other message. For $x, y \in \mathtt{M}^*$, we define $\#_C(x, y)$ to be the size of the set $\{i|\ x(i) = y(i)\}$. In other words, $\#_C(x, y)$ is the number of elements which must change in order to obtain y from x. We define $P_C(x, y) = \left(\frac{\lambda_C}{|\mathtt{M}|-1}\right)^{\#_C(x,y)} \cdot (1 - \lambda_C)^{|x|-\#_C(x,y)}$ if $|x| = |y|$, and $P_C(x, y) = 0$ otherwise. We extend $P_C(x, y)$ to $\mathtt{S} \times (\mathtt{C} \mapsto \mathtt{M}^*)$ as before. This induces a Markov chain in a similar manner to the case with duplication.

Lemma 12. *For each λ, w, λ_C, and PLCS $(\mathtt{S}, \mathtt{C}, \mathtt{M}, \mathtt{T}, \lambda, w, \lambda_C)$, the set $\{(\mathtt{s}, \mathtt{w})\ |\ |\mathtt{w}| = 0\}$ is an attractor.*

From Lemma 11 and Lemma 12 we can derive in a similar manner to Section 5.

Theorem 3. *Probabilistic Reachability and Probabilistic Repeated Reachability are decidable for PLCS with corruption errors.*

Other Unreliability Sources In a similar manner to the cases with duplication and corruption, we can obtain decidability results for models involving other

sources of unreliability such as insertion of messages [CFI96]. Furthermore, we can combine different sources of unreliability. For instance, we can consider models where we have both duplication and corruption together with lossiness. The crucial aspect of the model is that unreliability sources which may increase the number of messages inside the channels (such as insertion and duplication but not corruption) should have sufficiently low probabilities (compared to lossiness) to guarantee existence of a finite attractor.

7 Automata Definable Properties

In this section we consider more general properties than reachability and repeated reachability for PLCS. Let φ be a property of computations. We will be interested in whether

$$Prob\,\{\pi|\ \pi \text{ is a computation from } s \text{ in PLCS } M \text{ and } \pi \text{ satisfies } \varphi\ \} = 1.$$

We show that if the properties of computations are specified by (the ω-behavior of) finite state automata or equivalently by formulas of Monadic Logic of Order then the above problem is decidable

In order to check a property defined by a finite state automaton, we take its product with the given PLCS. The acceptance conditions are reduced to the reachability problem for the non-probabilistic system underlying the product. Similar results hold for the faulty probabilistic systems considered in section 6. The proofs for these systems follow the same pattern as for PLCS, therefore here we will confine ourself only with PLCS.

We consider an extension of LCS by adding a labeling function. A *state labeled* LCS is an LCS together with a finite alphabet Σ and a labeling function lab from the local states to Σ. Throughout this section we always assume that LCS are state labeled and will often use "LCS" for "state labeled LCS". We lift the labeling from LCS to the *state labeled transition system* $T = (S, \longrightarrow, \Sigma, lab)$ induced by an LCS $\mathcal{L}$: the label of every state in T is the same as the label of its local state component. Similarly, with a path $s_0, s_1, \ldots$ we associate an ω-string $lab(s_1), lab(s_2), \ldots$ over the alphabet Σ. When we deal here with probabilistic lossy channel systems we also assume that the underlying LCS is labeled, and this labeling is lifted to the labeling of the corresponding Markov chain. In this manner we obtain *state labeled PLCS* inducing *state labeled Markov chains*.

Next, we recall basic definitions and notations about finite state automata and cite a classical theorem (Theorem 4 [Tho90]) that automata have the same expressive power as monadic logic of order. A *finite automaton* $\mathcal{A}$ is a tuple $(\mathcal{Q},\ \Sigma,\ \rightarrow,\ q_0, \mathcal{F})$, consisting of a finite set $\mathcal{Q}$ of *states*, a finite alphabet Σ of *actions*, a *transition relation* $\rightarrow$ which is a subset of $\mathcal{Q} \times \Sigma \times \mathcal{Q}$, $q_0 \in \mathcal{Q}$ is the *initial state* of $\mathcal{A}$, and $\mathcal{F} \subseteq 2^{\mathcal{Q}}$ is a collection of *fairness conditions*. We write $q \xrightarrow{a} q'$ if $\langle q,\ a,\ q' \rangle \in \rightarrow$.

A *run* of $\mathcal{A}$ is an ω-sequence $q_0 a_0 q_1 a_1 \ldots$ such that $q_i \xrightarrow{a_i} q_{i+1}$ for all i. Such a run meets the the fairness conditions if the set of states that occur in the run

infinitely many times is a member of $\mathcal{F}$. An ω-string $a_0, a_1 \ldots$ over Σ is accepted by $\mathcal{A}$ if there is a run $q_0 a_0 q_1 a_1 \ldots$ that meets the fairness conditions of $\mathcal{A}$. The ω-language *accepted* by $\mathcal{A}$ is the set of all ω-strings acceptable by $\mathcal{A}$. We say that $\mathcal{A}$ is *deterministic* if for every state q and every letter b there is a unique q' such that $q \xrightarrow{b} q'$.

Theorem 4. *The following conditions are equivalent for ω-language L:*

1. *L is acceptable by a finite state automaton.*
2. *L is acceptable by a deterministic finite state automaton.*
3. *L is definable by a monadic formula .*

Products We define products of automata and state labeled transition systems. We also define products of automata and state labeled Markov chains. We investigate the reachability problem for these products and provide reduction of verification of automata definable properties of computations to the reachability problem. Consider an automaton $\mathcal{A} = (\mathcal{Q},\ \Sigma,\ \rightarrow,\ q_0, \mathcal{F})$, and a state labeled transition system $T = (S, \longrightarrow, \Sigma, lab)$. The *product* of $\mathcal{A}$ and M is a state labeled transition system defined as follows:

States: $Q \times S$ - the Cartesian product of the states of $\mathcal{A}$ and of T.
Labeling: A state (q, s) is labeled by $lab(s)$, i.e., it has the same label as s in T.
Transition relation: There is transition from (q, s) to (q', s') iff there is a transition $q \xrightarrow{lab(s)} q'$ in $\mathcal{A}$ and there is a transition from s to s' in T.

Problem 1
Instance A state labeled LCS which defines a state labeled transition system $T = (S, \rightarrow,\ lab, \Sigma)$, an automaton $\mathcal{A}$, states s_1 and s_2 in the product of T and $\mathcal{A}$.
Question Is s_2 reachable from s_1?

Problem 2
Instance A state labeled LCS which defines a state labeled transition system $T = (S, \rightarrow,\ lab, \Sigma)$, an automaton $\mathcal{A}$, states s_1 and a finite set of states S_2 in the product of T and $\mathcal{A}$.
Question Is the upward closure of S_2 reachable from s_1?

Lemma 13. *Problem 1 and Problem 2 are decidable.*

Next, we consider products of automata and state labeled Markov chains. Consider a deterministic automaton $\mathcal{A} = \langle Q,\ \Sigma,\ \rightarrow,\ q_0,\ \mathcal{F} \rangle$ and a state labeled Markov chain $M = (S, P,\ lab,\ \Sigma)$. The product of A and M is a state labeled Markov chain defined as follows:

States: $Q \times S$ - the Cartesian product of the states of $\mathcal{A}$ and of M.
Labeling: A state (q, s) is labeled by $lab(s)$, i.e., it has the same label as s in M.
Transition relation: The probability of transition from (q, s) to (q', s') is p iff there is a transition $q \xrightarrow{lab(s)} q'$ in $\mathcal{A}$ and the probability of transition from s to s' in M is p.

Observe that the requirement that $\mathcal{A}$ is deterministic ensures that the sum of probabilities of the transitions from the state (q, s) is the same as the sum of probabilities of the transitions from the state s in M, i.e. the sum is one. Hence the product is indeed a labeled Markov chain.

We say that a computation $s_1, s_2, \ldots$ is accepted by an automaton iff the corresponding ω-string $lab(s_1), lab(s_2), \ldots$ is accepted

Lemma 14. *Let $\mathcal{A}$ be a deterministic automaton with a set $\mathcal{F}$ of fairness conditions, let M be a labeled Markov chain, let R be the product of $\mathcal{A}$ and M, and let B be an attractor of R. Then the following are equivalent:*

1. *The probability of the set computations of M that start at s and are accepted by $\mathcal{A}$ is one.*
2. *For each BSCC C in $Graph(B)$, if C is reachable from s then there is F in $\mathcal{F}$ such that*
   (a) if (q, u) is reachable from C in R then $q \in F$ and
   (b) for each $q \in F$ there is $u \in M$ such that (q, u) is reachable from C in R.

Probabilistic Model Checking The next problem deals with probabilistic LCS.
Problem: Probabilistic Model-checking.
Instance A stated labeled PLCS which defines a state labeled Markov chain M, a state s in M, and an automaton $\mathcal{A}$.
Question Is the probability that a computation of M that starts at s is accepted by $\mathcal{A}$ equal to one?

Theorem 5. *Probabilistic Model-checking. Problem is decidable.*

Proof. Let R be the product of $\mathcal{A}$ and M. It is easy to see, by the same arguments as in Lemma 7, that the set B of states with empty channels in R is a finite attractor for R. By Lemma 13, we obtain that $Graph(B)$ is computable. Now, applying reachability algorithm of Lemma 13, we can verify the conditions of Lemma 14(2). By Lemma 14 these conditions are satisfied if and only if the probability that a computation of M that starts at s is accepted by $\mathcal{A}$ equal to one.

8 Conclusions and Discussion

We have shown decidability of model checking for a realistic class of probabilistic lossy channel systems, where during each step of the runs of the systems, any message inside the channels may be lost with a certain predefined probability.

In Section 5 we assume that our LCS are deadlock-free. In case of existence of deadlock states, Lemma 7 does not hold. However, it is straightforward to modify our algorithm to deal with deadlock. This follows from the fact that, we can use the reachability algorithm in [AJ96b] in order to check reachability of deadlock states.

A work closely related to this is [BE99]. In fact, our work can been as a generalization of the ideas presented in [BE99]. More precisely, in [BE99], a

model of PLCS is considered where at each state either one message lost or an non-lossy transition is performed. The probability λ of losing messages is assumed to be at least 0.5. Under this semantics, it is proved that for an PLCS the set $\{(\mathtt{s},\mathtt{w}) \mid |\mathtt{w}| = 0\}$ is an attractor. The decidability of reachability follows then in a similar manner to Section 5. Also, in [BE99], in contrast to the model of LCS presented in this paper, loss transitions are explicit. Therefore, the product of the transition system generated by an LCS with an automaton (Section 7), might not be equivalent to the transition system generated by any other LCS. In fact, under this semantics, it is undecidable whether the set of computations of a PLCS is accepted by a finite state automaton with probability one. To overcome this difficulty, it is assumed in [BE99] that the given automaton accepts an ω-language which is closed under stuttering.

Acknowledgement The work of the first author was partially supported by the European project ADVANCE, Contract No. IST – 1999-29082.

References

[ABIJ00] Parosh Aziz Abdulla, Christel Baier, Purushothaman Iyer, and Bengt Jonsson. Reasoning about probabilistic lossy channel systems. In C. Palamidessi, editor, *Proc. CONCUR 2000,* 11^{th} *Int. Conf. on Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, 2000.

[AJ96a] Parosh Aziz Abdulla and Bengt Jonsson. Undecidable verification problems for programs with unreliable channels. *Information and Computation*, 130(1):71–90, 1996.

[AJ96b] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.

[BE99] C. Baier and B. Engelen. Establishing qualitative properties for probabilistic lossy channel systems. In Katoen, editor, *ARTS'99, Formal Methods for Real-Time and Probabilistic Systems, 5th Int. AMAST Workshop*, volume 1601 of *Lecture Notes in Computer Science*, pages 34–52. Springer Verlag, 1999.

[Boc78] G. V. Bochman. Finite state description of communicating protocols. *Computer Networks*, 2:361–371, 1978.

[BS03] N. Bertrand and Ph. Schnoebelen. Model checking lossy channels systems is probably decidable. In *Proc. FOSSACS03, Conf. on Foundations of Software Science and Computation Structures*, 2003.

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 2(5):323–342, April 1983.

[CFI96] Gérard Cécé, Alain Finkel, and S. Purushothaman Iyer. Unreliable channels are easier to verify than perfect channels. *Information and Computation*, 124(1):20–31, 10 January 1996.

[Hig52] G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc.*, 2:326–336, 1952.

[KSK66] J.G. Kemeny, J.L. Snell, and A.W. Knapp. *Denumerable Markov Chains.* D Van Nostad Co., 1966.

[Tho90] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Methods and Semantics*, pages 133–192, 1990.

Generalized Iteration and Coiteration for Higher-Order Nested Datatypes

Andreas Abel[1⋆], Ralph Matthes[2], and Tarmo Uustalu[3⋆⋆]

[1] Department of Computer Science, University of Munich
abel@informatik.uni-muenchen.de
[2] Preuves, Programmes et Systèmes,
CNRS, Université Paris VII (on leave from University of Munich)
matthes@informatik.uni-muenchen.de
[3] Inst. of Cybernetics, Tallinn Technical University
tarmo@cs.ioc.ee

Abstract. We solve the problem of extending Bird and Paterson's generalized folds for nested datatypes and its dual to inductive and coinductive constructors of arbitrarily high ranks by appropriately generalizing Mendler-style (co)iteration. Characteristically to Mendler-style schemes of disciplined (co)recursion, the schemes we propose do not rest on notions like positivity or monotonicity of a constructor and facilitate programming in a natural and elegant style close to programming with the customary `letrec` construct, where the typings of the schemes, however, guarantee termination. For rank 2, a smoothened version of Bird and Paterson's generalized folds and its dual are achieved; for rank 1, the schemes instantiate to Mendler's original (re)formulation of iteration and coiteration. Several examples demonstrate the power of the approach. Strong normalization of our proposed extension of system F^ω of higher-order parametric polymorphism is proven by a reduction-preserving embedding into pure F^ω.

1 Introduction

Within the paradigm of generic programming, Bird and Paterson [8] with colleagues [11,15] have studied the problem of identifying workable schemes for defining functions for *nested*, non-uniform or heterogeneous *datatypes*, i.e., inductive and coinductive constructors of rank 2 (type transformers), and put forth *generalized folds* as a scheme for defining functions like substitution for the de Bruijn notation of lambda terms in a natural fashion.

In [2], two of the authors of the present article showed that, making good use of right notions of containment and monotonicity of constructors, the schemes of

⋆ The first author gratefully acknowledges the support by the PhD Programme *Logic in Computer Science* (GKLI) of the *Deutsche Forschungs-Gemeinschaft*.

⋆⋆ The third author is partially supported by the Estonian Science Foundation (ETF) under grant No. 4155. He is also grateful to the GKLI for two invitations to Munich; the cooperation started during these visits.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 54–69, 2003.

iteration and *coiteration* are extensible to monotone (co)inductive constructors of any finite kind. In the present article, we similarly extend the more liberal generalized folds to all finite kinds. We accomplish this thanks to two ideas: a simple, but powerful generalization of the notion of constructor containment, and reformulation of the schemes in the style originated by Mendler [18]. The result is a concise extension of system F^ω of higher-order parametric polymorphism with (co)inductive constructors of any finite kind, equipped with Mendler-style generalized (co)iteration. Switching to Mendler style was not intentional, but in the end turned out rewarding. The reasons are the following.

Firstly, any syntactic positivity requirement can be avoided in the formation rules of (co)inductive types. This is beneficial as positivity and map terms associating to positive constructors would have to be defined by induction outside the system and parametrically polymorphic quantification over all positive constructors is impossible. Moreover, for higher kinds, there is no obvious canonical definition of positivity, although attempts of definition exist [14]. Replacing positivity with monotonicity [17,2] gives an improvement, but formulations of the systems and especially programming remain clumsy.

Secondly, Mendler style facilitates a programming style very close to programming with general recursion (i.e., the `letrec` construct). The computation rules for Mendler-style disciplined (co)recursion schemes are nearly identical to the rule of `letrec`, the restrictive typings however ensure that all computations terminate.

Thirdly, Mendler-style disciplined (co)recursion schemes tend to be amenable for generalizations whereas conventional ones—making use of map terms or monotonicity witnesses—typically get complicated. Examples are: primitive (co-) recursion [18], course-of-value (co)iteration [21,22], iteration over multiple inductive types at the same time [22] and—as the examples of this article testify—generalized iteration in the sense of generalized folds.

The article is organized as follows. In Sect. 2, we review our starting point system F^ω of higher-order parametric polymorphism. In Sect. 3, we present our system MIt^ω of (co)inductive constructors of finite ranks with generalized Mendler-style iteration and describe some programming examples. The embedding of MIt^ω into F^ω is presented in Sect. 4. We conclude with a summary and discussion of related work.

Acknowledgements: Many thanks to Peter Hancock for his suggestion in November 2000 of the unusual notion $F \leq_{\kappa 1} G$. It started this whole research project.

2 System F^ω

Our development of higher-order datatypes takes place within a conservative extension of Curry-style system F^ω by binary sums and products, the unit type and existential quantification. It contains three syntactic categories:

Kinds. Kinds are given by the following grammar and denoted by the letter κ.

$$\begin{array}{lll} \kappa & ::= & * \mid \kappa \to \kappa' \\ \mathsf{rk}(*) & := & 0 \\ \mathsf{rk}(\kappa \to \kappa') & := & \max(\mathsf{rk}(\kappa)+1, \mathsf{rk}(\kappa')) \end{array}$$

The *rank* of kind κ is computed by $\mathsf{rk}(\kappa)$. We introduce abbreviations for some special kinds: $\kappa 0 = *$, *types*, $\kappa 1 = * \to *$, *unary type transformers* and $\kappa 2 = (* \to *) \to * \to *$ *unary transformers of type transformers*.

Note that each kind κ' can be uniquely written as $\boldsymbol{\kappa} \to *$, where we write $\boldsymbol{\kappa}$ for the sequence $\kappa_1, \ldots, \kappa_n$ and set $\boldsymbol{\kappa} \to \kappa := \kappa_1 \to \ldots \to \kappa_n \to \kappa$. Provided another sequence $\boldsymbol{\kappa}' = \kappa'_1, \ldots, \kappa'_n$ of the same length, i.e., $|\boldsymbol{\kappa}'| = |\boldsymbol{\kappa}|$, set $\boldsymbol{\kappa} \to \boldsymbol{\kappa}' := \kappa_1 \to \kappa'_1, \ldots, \kappa_n \to \kappa'_n$. This last abbreviation does not conflict with the abbreviation $\boldsymbol{\kappa} \to \kappa$ due to the required $|\boldsymbol{\kappa}'| = |\boldsymbol{\kappa}|$.

Constructors. Uppercase latin letters denote constructors, given by the following grammar. The metavariable X ranges over a denumerable set of constructor variables.

$$\begin{array}{rl} A, B, F, G ::= & X \mid \lambda X^\kappa.F \mid F\,G \mid \forall F^\kappa.\,A \mid \exists F^\kappa.\,A \mid A \to B \\ & \mid A + B \mid A \times B \mid 1 \end{array}$$

We identify β-equivalent constructors. A constructor F has kind κ if there is a context Γ such that $\Gamma \vdash F : \kappa$. The kinding rules for constructors appear in Appendix A.

The rank of a constructor is given by the rank of its kind. Preferably we will use letters A, B, C, D for constructors of rank 0 *(types)* and F, G, H for constructors of rank 1. If no kinds are given and cannot be guessed from the context, we assume $A, B, C, D : *$ and $F, G, H : \kappa 1$. Write $\mathsf{Id}_\kappa := \lambda X^\kappa.X$ for the identity constructor. If the kinding is clear from the context, we just write Id. Constructor application associates to the left, i. e., $F\,G_1\,\ldots\,G_n = (\ldots(F\,G_1)\,\ldots)\,G_n$. Setting $\boldsymbol{G} := G_1, \ldots, G_n$, the constructor $F\,G_1\,\ldots\,G_n$ is also written as $F\boldsymbol{G}$. Sums and products can inductively be extended to all kinds: For $F, G : \kappa_1 \to \kappa_2$ set $F + G := \lambda X^{\kappa_1}.FX + GX$ and $F \times G := \lambda X^{\kappa_1}.FX \times GX$.

Objects (Curry terms). Lower case letters denote terms. In the grammar below, the metavariable x ranges over a denumerable set of object variables.

$$\begin{array}{rl} r, s, t ::= & x \mid \lambda x.t \mid r\,s \mid \mathsf{inl}\,t \mid \mathsf{inr}\,t \mid \mathsf{case}\,(r,\, x.\,s,\, y.\,t) \\ & \mid \langle\rangle \mid \langle t_0, t_1\rangle \mid r.0 \mid r.1 \mid \mathsf{pack}\,t \mid \mathsf{open}\,(r,\, x.\,s) \end{array}$$

Most term constructors are standard; "pack" introduces and "open" eliminates existential quantification. The polymorphic identity $\lambda x.x : \forall A.\,A \to A$ will be denoted by id. We write $f \circ g$ for function composition $\lambda x.\,f\,(g\,x)$. Application $r\,s$ associates to the left, hence $r\boldsymbol{s} = (\ldots(r\,s_1)\,\ldots\,s_n)$ for $\boldsymbol{s} = s_1, \ldots, s_n$.

A term t has type A if $\Gamma \vdash t : A$ for some context Γ. The relation $\longrightarrow$ denotes the usual one-step β-reduction which is confluent, type preserving and strongly normalizing. The typing and reduction rules for terms are standard and can be found in Appendix A.

In the following we will refer to the here defined system simply as "F^ω".

3 Generalized Mendler-Style Iteration and Coiteration

In this section, we recap and extend the notions of containment and monotonicity presented in [2]. On top of these notions, we define the system MIt^ω of generalized (co)iteration for inductive and coinductive constructors of arbitrary ranks, which we then specialize to rank 1 (types) and rank 2 (type transformers). To give a feel for our system, we spell out some examples involving *nested* or non-uniform *datatypes* [7].

3.1 Containment and Monotonicity of Constructors

Containment. The key to extending Mendler-style iteration and coiteration [19] to finite kinds consists in identifying an appropriate containment relation for constructors of the same kind κ. For types, the canonical choice is implication. For an arbitrary kind $\kappa = \boldsymbol{\kappa} \to *$, the easiest notion is "pointwise implication": The constructor $\subseteq_\kappa\colon \kappa \to \kappa \to *$ is defined by $F \subseteq_\kappa G := \forall \boldsymbol{X}^{\boldsymbol{\kappa}}.\, F\boldsymbol{X} \to G\boldsymbol{X}$, hence $F \subseteq_\kappa G$ is a type which, as a proposition, states that F is contained in G.

A more refined notion $\leq_\kappa$ has been employed already in previous work [2] which studies (co)iteration for monotone (co)inductive constructors of higher kinds:

$$\begin{array}{lcl} F \leq_* G & := & F \to G \\ F \leq_{\kappa\to\kappa'} G & := & \forall X^\kappa \forall Y^\kappa.\, X \leq_\kappa Y \to F\,X \leq_{\kappa'} G\,Y \end{array}$$

Monotonicity. Using this notion of containment, we can define monotonicity $\mathsf{mon}_\kappa : \kappa \to *$ for kind κ directly by

$$\mathsf{mon}_\kappa\, F := F \leq_\kappa F.$$

The type $\mathsf{mon}_\kappa\, F$, seen as a proposition, asserts that F is monotone. The same type is used in polytypic programming for generic map functions [13,3].

This notion does not enter the formulation of system MIt^ω, but many applications. We omit the subscripted kind κ when clear from the context, as in the definition of the following basic *monotonicity witnesses*. These are closed terms whose type is some $\mathsf{mon}\, F$. They will pop up in examples later.

$$\begin{array}{lll} \mathsf{pair} & : \mathsf{mon}(\lambda A \lambda B.A \times B) & := \lambda f \lambda g \lambda p.\, \langle f\,(p.0),\, g\,(p.1)\rangle \\ \mathsf{fork} & : \mathsf{mon}(\lambda A.A \times A) & := \lambda f.\, \mathsf{pair}\, f\, f \\ \mathsf{either} & : \mathsf{mon}(\lambda A \lambda B.A + B) & := \lambda f \lambda g \lambda x.\, \mathsf{case}\,(x,\, a.\, \mathsf{inl}\,(f\,a),\, b.\, \mathsf{inr}\,(g\,b)) \\ \mathsf{maybe} & : \mathsf{mon}(\lambda A.1 + A) & := \mathsf{either}\ \mathsf{id} \end{array}$$

Relativized refined containment. In order to be able to extend Mendler (co)iteration to higher kinds so that generalized folds [8] are covered, we have to relativize the notion $\leq_\kappa$, $\kappa = \boldsymbol{\kappa} \to *$, to a vector $\boldsymbol{H}$ of constructors of kinds $\boldsymbol{\kappa} \to \boldsymbol{\kappa}$. For every kind $\kappa = \boldsymbol{\kappa} \to *$, we define a constructor $\leq_\kappa^{(-)}\colon (\boldsymbol{\kappa} \to \boldsymbol{\kappa}) \to \kappa \to \kappa \to *$ by structural recursion on κ as follows:

$$\begin{array}{lcl} F \leq_*^{\boldsymbol{H}} G & := & F \to G \\ F \leq_{\kappa\to\kappa'}^{H,\boldsymbol{H}} G & := & \forall X^\kappa \forall Y^\kappa.\, X \leq_\kappa H\,Y \to F\,X \leq_{\kappa'}^{\boldsymbol{H}} G\,Y \end{array}$$

Note that, in the second line, H has kind $\kappa \to \kappa$. For $\boldsymbol{H}$ a vector of identity constructors, the new notion $\leq_\kappa^{\boldsymbol{H}}$ coincides with $\leq_\kappa$. Similarly, we define another constructor ${}^{(-)}\leq_\kappa\colon (\boldsymbol{\kappa} \to \boldsymbol{\kappa}) \to \kappa \to \kappa \to *$, where the base case is the same as before, hence no ambiguity with the notation arises.

$$\begin{array}{lcl} F \quad \leq_* \quad G & := & F \to G \\ F \;\; {}^{H,\boldsymbol{H}}\!\leq_{\kappa\to\kappa'} \; G & := & \forall X^\kappa \forall Y^\kappa.\, H\,X \leq_\kappa Y \to F\,X \;\; {}^{\boldsymbol{H}}\!\leq_{\kappa'} G\,Y \end{array}$$

As an example, for $F, G, H : \kappa 1$, one has

$$\begin{array}{lcl} F \quad \leq_{\kappa 1}^{H} \quad G & = & \forall A \forall B.\, (A \to HB) \to FA \to GB, \\ F \;\; {}^{H}\!\leq_{\kappa 1} \; G & = & \forall A \forall B.\, (HA \to B) \to FA \to GB. \end{array}$$

3.2 System MIt^ω

Now we are ready to define generalized Mendler-style iteration and coiteration, which specialize to ordinary Mendler-style iteration and coiteration in the case of (co)inductive types, and to a scheme encompassing generalized folds [8,11,15] and the dual scheme for coinductive constructors of rank 2. This gives an extension of Mendler's system [19] to finite kinds. The generalized scheme for coinductive constructors is a new principle of programming with non-wellfounded datatypes.

The system MIt^ω is given as an extension of F^ω by wellkinded constructor constants μ_κ and ν_κ, welltyped term constants $\mathsf{in}_\kappa, \mathsf{GIt}_\kappa, \mathsf{out}_\kappa$ and GCoit_κ for every kind κ, and new term reduction rules.

Inductive constructors. Let $\kappa = \boldsymbol{\kappa} \to *$ and $\boldsymbol{\kappa}' = \boldsymbol{\kappa} \to \boldsymbol{\kappa}$.

$$\begin{array}{lll} \text{(Form)} & \mu_\kappa & : (\kappa \to \kappa) \to \kappa \\ \text{(Intro)} & \mathsf{in}_\kappa & : \forall F^{\kappa\to\kappa}.\, F\,(\mu_\kappa F) \subseteq_\kappa \mu_\kappa F \\ \text{(Elim)} & \mathsf{GIt}_\kappa & : \forall F^{\kappa\to\kappa} \forall \boldsymbol{H}^{\boldsymbol{\kappa}'} \forall G^\kappa.\, (\forall X^\kappa.\, X \leq^{\boldsymbol{H}} G \to F\,X \leq^{\boldsymbol{H}} G) \to \mu_\kappa F \leq^{\boldsymbol{H}} G \\ \text{(Red)} & \mathsf{GIt}_\kappa\; s\; \boldsymbol{f}\; (\mathsf{in}_\kappa\; t) & \longrightarrow_\beta s\; (\mathsf{GIt}_\kappa\; s)\; \boldsymbol{f}\; t \end{array}$$

with $|\boldsymbol{f}| = |\boldsymbol{\kappa}|$.

Coinductive constructors. Let $\kappa = \boldsymbol{\kappa} \to *$ and $\boldsymbol{\kappa}' = \boldsymbol{\kappa} \to \boldsymbol{\kappa}$.

$$\begin{array}{lll} \text{(Form)} & \nu_\kappa & : (\kappa \to \kappa) \to \kappa \\ \text{(Elim)} & \mathsf{out}_\kappa & : \forall F^{\kappa\to\kappa}.\, \nu_\kappa F \subseteq_\kappa F\,(\nu_\kappa F) \\ \text{(Intro)} & \mathsf{GCoit}_\kappa & : \forall F^{\kappa\to\kappa} \forall \boldsymbol{H}^{\boldsymbol{\kappa}'} \forall G^\kappa.\, (\forall X^\kappa.\, G \;\; {}^{\boldsymbol{H}}\!\leq X \to G \;\; {}^{\boldsymbol{H}}\!\leq F\,X) \to G \;\; {}^{\boldsymbol{H}}\!\leq \nu_\kappa F \\ \text{(Red)} & \mathsf{out}_\kappa\; (\mathsf{GCoit}_\kappa\; s\; \boldsymbol{f}\; t) & \longrightarrow_\beta s\; (\mathsf{GCoit}_\kappa\; s)\; \boldsymbol{f}\; t \end{array}$$

with $|\boldsymbol{f}| = |\boldsymbol{\kappa}|$.

Notice that for *every* constructor F of kind $\kappa \to \kappa$, $\mu_\kappa F$ is a constructor of kind κ. In Mendler's original system [19] as well as its variant for the treatment of primitive (co-)recursion [18], always positivity of F is required which is a very natural concept in the case $\kappa = *$. However, for higher kinds, there does not

exist such a canonical syntactic restriction. Anyway, in [21] it has been observed that, in order to prove strong normalization, *there is no need for the restriction to positive inductive types*—an observation which has been the cornerstone for the treatment of monotone inductive types in [16] and becomes even more useful for our higher-order nested datatypes.

As for F^ω, denote the term closure of the reduction rules by $\longrightarrow$ and its transitive closure by $\longrightarrow^+$.

3.3 Mendler-Style (Co)Iteration for (Co)Inductive Types

In the case $\kappa = *$, the rules for μ_κ and ν_κ match with Mendler's [19], except for our removal of the positivity condition and our choice of Curry-style typing:

Inductive types.

$$\begin{array}{lll}
\text{(Form)} & \mu_* & : (* \to *) \to * \\
\text{(Intro)} & \mathsf{in}_* & : \forall F^{*\to *}.\, F\,(\mu_* F) \to \mu_* F \\
\text{(Elim)} & \mathsf{GIt}_* & : \forall F^{*\to *} \forall Y^*.\, (\forall X^*.\, (X \to Y) \to F\,X \to Y) \to \mu_* F \to Y \\
\text{(Red)} & \mathsf{GIt}_*\, s\,(\mathsf{in}_*\, t) & \longrightarrow_\beta s\,(\mathsf{GIt}_*\, s)\, t
\end{array}$$

Coinductive types.

$$\begin{array}{lll}
\text{(Form)} & \nu_* & : (* \to *) \to * \\
\text{(Elim)} & \mathsf{out}_* & : \forall F^{*\to *}.\, \nu_* F \to F\,(\nu_* F) \\
\text{(Intro)} & \mathsf{GCoit}_* & : \forall F^{*\to *} \forall Y^*.\, (\forall X^*.\, (Y \to X) \to Y \to F\,X) \to Y \to \nu_* F \\
\text{(Red)} & \mathsf{out}_*\,(\mathsf{GCoit}_*\, s\, t) & \longrightarrow_\beta s\,(\mathsf{GCoit}_*\, s)\, t
\end{array}$$

Relation to general recursion. Typed functional programming languages like ML and Haskell use recursive types instead of inductive and coinductive types and general recursion instead of strongly normalizing restrictions such as Mendler (co)iteration. General recursion can be introduced via a fixed-point combinator

$$\begin{aligned}
&\mathsf{fix} : \forall A.(A \to A) \to A \\
&\mathsf{fix}\ s \longrightarrow s\,(\mathsf{fix}\ s),
\end{aligned}$$

from which the more common `let rec` $f = r$ `in` t can be defined as `let` $f =$ $\mathsf{fix}\ (\lambda f.\, r)$ `in` t. A nice aspect of Mendler (co)iteration is that the reduction behaviour GIt_* and GCoit_* is almost identical to the one of fix. The only difference is that unfolding of GIt resp. GCoit is controlled by a guard "in" resp. "out", which gets removed in the reduction step. Guarded unfolding of recursion is essential to strong normalization; similar setups can be found in other systems which facilitate *type-based termination*, e.g. [10,1,5].

In some sense GIt and GCoit are just restricted versions of fix, i. e., each rank-1 MIt^ω program translates (requiring minimal changes) into a Haskell program with the same meaning. For higher kinds κ, GIt_κ and GCoit_κ are not typable

in the Hindley-Milner type systems of Haskell 98 and ML, but their reduction behaviour is still included in the one of fix. This suggests that one can code most naturally with GIt and GCoit, which we will demonstrate in the next subsection by some examples involving so-called *nested* or *heterogeneous datatypes*.

3.4 Programming with (Co)Inductive Constructors of Rank 2

Nested or non-uniform datatypes, i.e., inductive and coinductive constructors of rank 2 (more exactly, inductive and coinductive constructors induced by constructors of rank 2), arise in our system as applications of $\mu_{\kappa 1}$ and $\nu_{\kappa 1}$ (recall that $\kappa 1 = * \to *$ and $\kappa 2 = \kappa 1 \to \kappa 1$). We obtain the following instances from the general definitions.

Inductive constructors of rank 2.

$$\begin{array}{ll}
\text{(Form)} & \mu_{\kappa 1} \quad : \kappa 2 \to \kappa 1 \\
\text{(Intro)} & \mathsf{in}_{\kappa 1} \quad : \forall F^{\kappa 2} \forall A.\, F\,(\mu_{\kappa 1} F)\,A \to \mu_{\kappa 1} F A \\
\text{(Elim)} & \mathsf{GIt}_{\kappa 1} : \forall F^{\kappa 2} \forall H^{\kappa 1} \forall G^{\kappa 1}.\ (\forall X^{\kappa 1}.\, X \leq^H G \to F\,X \leq^H G) \to \mu_{\kappa 1} F \leq^H G \\
\text{(Red)} & \mathsf{GIt}_{\kappa 1}\, s\, f\, (\mathsf{in}_{\kappa 1}\, t) \longrightarrow_\beta s\, (\mathsf{GIt}_{\kappa 1}\, s)\, f\, t
\end{array}$$

Coinductive constructors of rank 2.

$$\begin{array}{ll}
\text{(Form)} & \nu_{\kappa 1} \quad : \kappa 2 \to \kappa 1 \\
\text{(Elim)} & \mathsf{out}_{\kappa 1} \quad : \forall F^{\kappa 2} \forall A.\, \nu_{\kappa 1} F A \to F\,(\nu_{\kappa 1} F) A \\
\text{(Intro)} & \mathsf{GCoit}_\kappa : \forall F^{\kappa 2} \forall H^{\kappa 1} \forall G^{\kappa 1}.\ (\forall X^{\kappa 1}.\, G\ {}^H\!\leq X \to G\ {}^H\!\leq F\,X) \to G\ {}^H\!\leq \nu_{\kappa 1} F \\
\text{(Red)} & \mathsf{out}_{\kappa 1}\, (\mathsf{GCoit}_{\kappa 1}\, s\, f\, t) \longrightarrow_\beta s\, (\mathsf{GCoit}_{\kappa 1}\, s)\, f\, t
\end{array}$$

An example of a structure which can be modeled by a nested datatype is lists of length 2^n, which are called *powerlists* [6] or *perfectly balanced, binary leaf trees* [11]. In our system, they are represented by the type transformer $\mathsf{PList} := \mu_{\kappa 1}\, \mathsf{PListF}$ where $\mathsf{PListF} : \kappa 2 := \lambda F \lambda A.\, A + F(A \times A)$. The data constructors are given by

$$\begin{array}{ll}
\mathsf{zero} : \forall A.\, A \to \mathsf{PList}\, A & := \lambda a.\, \mathsf{in}_{\kappa 1}(\mathsf{inl}\, a) \\
\mathsf{succ} : \forall A.\, \mathsf{PList}(A \times A) \to \mathsf{PList}\, A & := \lambda l.\, \mathsf{in}_{\kappa 1}(\mathsf{inr}\, l)
\end{array}$$

Assume a type Nat of natural numbers with addition "+" and multiplication "×", both written infix. Suppose we want to define a function $\mathsf{sum} : \mathsf{PList}\,\mathsf{Nat} \to \mathsf{Nat}$ which sums up all elements of a powerlist by iteration over its structure. The case $\mathsf{sum}\,(\mathsf{succ}\ t)$ imposes some challenge, since sum cannot be directly applied to $t : \mathsf{PList}(\mathsf{Nat} \times \mathsf{Nat})$. The solution is to define a more general function sum' by polymorphic recursion, which has the following behaviour.

$$\begin{array}{l}
\mathsf{sum}' : \forall A.\, (A \to \mathsf{Nat}) \to \mathsf{PList}\, A \to \mathsf{Nat} \\
\mathsf{sum}'\ f\,(\mathsf{zero}\ a) \longrightarrow^+ f\, a \\
\mathsf{sum}'\ f\,(\mathsf{succ}\ l) \longrightarrow^+ \mathsf{sum}'\,(\lambda p.\, f\,(p.0) + f\,(p.1))\, l
\end{array}$$

Here, the iteration process builds up a continuation f which in the end sums up the contents packed into a. From sum', the summation function is obtained by $\mathsf{sum} := \mathsf{sum}'\ \mathsf{id}$.

The system MIt^ω has been designed so that functions like sum' can be defined directly via generalized iteration. In our case, use the instantiations $F := \mathsf{PListF}$ and $G := H := \lambda\_.\,\mathsf{Nat}$ and define:

$$\begin{array}{lcl} \mathsf{sum}' & : & \mu_{\kappa 1} F \leq^H G \\ \mathsf{sum}' & := & \mathsf{GIt}_{\kappa 1}\ \lambda sum'\lambda f\lambda x.\,\mathsf{case}\,(x,\ a.\,f\,a, \\ & & \qquad\qquad\qquad\qquad\qquad l.\,sum'\,(\lambda p.\,f\,(p.0) + f\,(p.1))\,l) \end{array}$$

The postulated reduction behaviour is verified by a simple calculation.

For another example consider the non-wellfounded version of perfectly balanced, binary (node-labelled) trees. They are represented by the type transformer $\mathsf{BTree} := \nu_{\kappa 1}\,\mathsf{BTreeF}$ where $\mathsf{BTreeF} : \kappa 2 := \lambda F\lambda A.\,A \times F(A \times A)$. The data destructors are

$$\begin{array}{ll} \mathsf{root} : \forall A.\ \mathsf{BTree}\,A \to A & := \lambda t.\,(\mathsf{out}_{\kappa 1} t).0 \\ \mathsf{subs} : \forall A.\ \mathsf{BTree}\,A \to \mathsf{BTree}(A \times A) & := \lambda t.\,(\mathsf{out}_{\kappa 1} t).1 \end{array}$$

We want to define the tree $\mathsf{nats} : \mathsf{BTree}\,\mathsf{Nat}$ filled with natural numbers starting with 1 breadth-first left-first. A more general function $\mathsf{nats}' : \forall A.\ (\mathsf{Nat} \to A) \to (\mathsf{Nat} \to \mathsf{BTree}\,A)$ with the reduction behaviour

$$\begin{array}{l} \mathsf{root}(\mathsf{nats}'\ f\,n) \longrightarrow^+ f\,n \\ \mathsf{subs}(\mathsf{nats}'\ f\,n) \longrightarrow^+ \mathsf{nats}'\,(\lambda m.\,\langle f\,(2 \times m), f\,(2 \times m + 1)\rangle)\,n \end{array}$$

is definable as a generalized coiteration by

$$\mathsf{nats}' := \mathsf{GCoit}_{\kappa 1}\ \lambda nats'\lambda f\lambda n.\ \langle f\,n,\ nats'\,(\lambda m.\,\langle f\,(2 \times m), f\,(2 \times m + 1)\rangle)\,n\rangle$$

choosing $F := \mathsf{BTreeF}$, $G := H := \lambda\_.\,\mathsf{Nat}$. To obtain nats, one sets $\mathsf{nats} := \mathsf{nats}'\ \mathsf{id}\ 1$.

Higher-order representation of de Bruijn terms. Bird & Paterson [9] and Altenkirch & Reus [4] have shown that nameless untyped λ-terms can be represented by a heterogeneous datatype. As in the system GMIC of [2], this type is obtained in MIt^ω as the least fixed point of the monotone rank-2 constructor LamF.

$$\begin{array}{ll} \mathsf{LamF} : \kappa 2 & := \lambda F\lambda A.\,A + (FA \times FA + F(1 + A)) \\ \mathsf{lamf}\ : \mathsf{mon}\,\mathsf{LamF} & := \lambda s\lambda f.\ \mathsf{either}\ f\ \Big(\mathsf{either}\,(\mathsf{fork}\,(s\,f))\,(s\,(\mathsf{maybe}\ f))\Big) \end{array}$$

The type $\mathsf{Lam}\,A$ again represents all de Bruijn terms with free variables in A, the constructors var, app and abs are simplified w. r. t. [2]. Again, we provide an auxiliary function weak which lifts each variable in a term to provide space for a fresh variable.

$$
\begin{array}{lll}
\mathsf{Lam} & : \kappa 1 & := \mu_{\kappa 1}\,\mathsf{LamF} \\
\mathsf{lam} & : \mathsf{mon}\,\mathsf{Lam} & := \mathsf{GIt}_{\kappa 1}\ \lambda map \lambda f \lambda x.\, \mathsf{in}_{\kappa 1}\,(\mathsf{lamf}\ \ map\, f\, x) \\
\mathsf{var} & : \forall A.\, A \to \mathsf{Lam}\, A & := \lambda a.\, \mathsf{in}_{\kappa 1}\,(\mathsf{inl}\, a) \\
\mathsf{app} & : \forall A.\, \mathsf{Lam}\, A \to \mathsf{Lam}\, A \to \mathsf{Lam}\, A & := \lambda t_1 \lambda t_2.\, \mathsf{in}_{\kappa 1}\,(\mathsf{inr}\,(\mathsf{inl}\,\langle t_1, t_2\rangle)) \\
\mathsf{abs} & : \forall A.\, \mathsf{Lam}(1 + A) \to \mathsf{Lam}\, A & := \lambda r.\, \mathsf{in}_{\kappa 1}\,(\mathsf{inr}\,(\mathsf{inr}\, r)) \\
\mathsf{weak} & : \forall A.\, \mathsf{Lam}\, A \to \mathsf{Lam}(1 + A) & := \mathsf{lam}\,(\lambda a.\ \mathsf{inr}\, a)
\end{array}
$$

The most natural question on this representation of untyped λ-calculus is the representability of substitution. With generalized iteration, it is possible to give a direct definition of substitution (the bind or extension operation of the lambda terms monad):

$$
\begin{array}{l}
\mathsf{subst} : \quad \forall A \forall B.\, (A \to \mathsf{Lam}\, B) \to \mathsf{Lam}\, A \to \mathsf{Lam}\, B \ \equiv\ \mathsf{Lam} \leq^{\mathsf{Lam}} \mathsf{Lam} \\
\mathsf{subst} := \mathsf{GIt}_{\kappa 1}\ \lambda subst \lambda f \lambda t.\ \mathsf{case}\,(t, \\
\qquad\qquad a.\ f\, a,\ t'.\ \mathsf{case}\,(t', \\
\qquad\qquad p.\ \mathsf{app}\,(subst\, f\,(p.0))\,(subst\, f\,(p.1)), \\
\qquad\qquad r.\ \mathsf{abs}\,(subst\,(\mathsf{lift}\, f)\, r)), \\
\mathsf{lift} \quad : \quad \forall A \forall B.\, (A \to \mathsf{Lam}\, B) \to (1 + A) \to \mathsf{Lam}\,(1 + B) \\
\mathsf{lift} \quad := \lambda f \lambda x.\ \mathsf{case}\,(x,\ u.\ \mathsf{var}\,(\mathsf{inl}\ u),\ a.\ \mathsf{weak}\,(f\, a))
\end{array}
$$

Note that the formulation of generalized folds in [8] would yield the flattening function (the join or multiplication operation of the monad)

$$\mathsf{flatten} : \forall A.\ \mathsf{Lam}(\mathsf{Lam}\, A) \to \mathsf{Lam}\, A.$$

We obtain flattening as special case of substitution by $\mathsf{flatten} := \mathsf{subst}\ \mathsf{id}$.

Triangles. The dual of substitution for variables in a term or non-wellfounded term is redecoration of a non-wellfounded or wellfounded decorated tree, cf. [23]. An interesting and intuitive example of decorated tree types arising from a rank-2 coinductive constructor are triangles. Define

$$
\begin{array}{ll}
\mathsf{TriF} := \lambda E \lambda F^{\kappa 1} \lambda A.\ A \times F(E \times A) & : \star \to \kappa 2 \\
\mathsf{Tri} \ := \lambda E.\ \nu_{\kappa 1}(\mathsf{TriF}\, E) & : \star \to \kappa 1
\end{array}
$$

Then $\mathsf{Tri}\, E A$ is the type of triangular tables of the sort

$$
\begin{array}{r|l}
A & E\ E\ E\ E\ \ldots \\
 & A\ E\ E\ E\ \ldots \\
 & \quad A\ E\ E\ \ldots \\
 & \qquad A\ E\ \ldots \\
 & \qquad\quad A\ \ldots
\end{array}
$$

decomposing into a scalar (an element of A) and a trapezium (an element of $\mathsf{Tri}\, E(E \times A)$). The destructors and the monotonicity witness are

$$
\begin{array}{lll}
\mathsf{top} := \lambda t.\ (\mathsf{out}_{\kappa 1}\, t).0 & : & \forall E \forall A.\, \mathsf{Tri}\, E A \to A \\
\mathsf{rest} := \lambda t.\ (\mathsf{out}_{\kappa 1}\, t).1 & : & \forall E \forall A.\, \mathsf{Tri}\, E A \to \mathsf{Tri}\, E(E \times A)
\end{array}
$$

$$\mathsf{tri} \ := \mathsf{GCoit}_{\kappa 1}\, \lambda map \lambda f \lambda x. \langle f(\mathsf{top}\ x), map(\mathsf{pair}\ \mathsf{id}\ f)(\mathsf{rest}\ x)\rangle : \forall E.\, \mathsf{mon}_{\kappa 1}(\mathsf{Tri}\, E)$$

Redecoration is an operation dual to substitution that takes a redecoration rule f (an assignment of B-decorations to A-decorated trees) and an A-decorated tree t, and returns a B-decorated tree t'. The return tree t' is obtained from t by B-redecorating every node based on the A-decorated subtree it roots, as instructed by the redecoration rule. For streams, for instance $\mathsf{redec} : \forall A \forall B.\, (\mathsf{Str}\, A \to B) \to \mathsf{Str}\, A \to \mathsf{Str}\, B)$ takes $f : \mathsf{Str}\, A \to B$ and $t : \mathsf{Str}\, A$ and returns $\mathsf{redec}\ \ f\ t$, which is a B-stream obtained from t by replacing each of its elements by what f assigns to the substream this element heads. Triangles are a generalization of streams much in the same way as de Bruijn notations for lambda terms differ from terms in the universal algebra style signature with one binary and one unary operator. For triangles, redecoration works as follows: In the triangle

$$\begin{array}{r} A\ E\ E\ E\ E\ \ldots \\ A\ E\ E\ E\ \ldots \\ \hline \underline{A}\ E\ E\ \ldots \\ A\ E\ \ldots \\ A\ \ldots \end{array}$$

the underlined A (as an example) gets replaced by the B assigned by the redecoration rule to the subtriangle cut out by the horizontal line; similarly, every other A is replaced by a B. This is straightforward to define using GCoit:

$$\begin{array}{lll} \mathsf{lift} & : & \forall E \forall A \forall B.\, (\mathsf{Tri}\, E A \to B) \to \mathsf{Tri}\, E (E \times A) \to E \times B \\ \mathsf{lift} & := & \lambda f \lambda y.\, \langle (\mathsf{top}\ y).0,\ f\, (\mathsf{tri}\, (\lambda p.\ p.1)\, y) \rangle \\ \mathsf{redec} & : & \forall E \forall A \forall B.\, (\mathsf{Tri}\, E A \to B) \to \mathsf{Tri}\, E A \to \mathsf{Tri}\, E B \\ \mathsf{redec} & := & \mathsf{GCoit}_{\kappa 1}\, \lambda redec \lambda f \lambda x.\, \langle f\, x, redec\, (\mathsf{lift}\, f)\, (\mathsf{rest}\, x) \rangle \end{array}$$

4 Embedding into System F^{ω}

In this section, we show how to embed MIt^{ω} into F^{ω}. The embedding establishes strong normalization for MIt^{ω}.

4.1 Kan Extensions

For the sake of the embedding of MIt^{ω} into its subsystem F^{ω}, we use a syntactic version of Kan extensions, see [20, chapter 10]. Compared with [2], Kan extensions "along" are now defined for all kinds, not just for rank 1.

Right Kan extension along $\boldsymbol{H}$. Let $\kappa = \boldsymbol{\kappa} \to *$ and $\boldsymbol{\kappa}' = \boldsymbol{\kappa} \to \boldsymbol{\kappa}$ and define for $G : \kappa$, $\boldsymbol{H} : \boldsymbol{\kappa}'$ and $\boldsymbol{X} : \boldsymbol{\kappa}$ the type $(\mathsf{Ran}_{\boldsymbol{H}}\, G)\, \boldsymbol{X}$ by iteration on $|\boldsymbol{\kappa}|$:

$$\begin{array}{rl} \mathsf{Ran}\ G & := G \\ (\mathsf{Ran}_{H,\boldsymbol{H}}\, G)\, X \boldsymbol{X} & := \forall Y^{\kappa_1}.\ X \leq HY\ \ \to\ \ (\mathsf{Ran}_{\boldsymbol{H}}\, (G\, Y))\, \boldsymbol{X} \end{array}$$

Left Kan Extension along $\boldsymbol{H}$. Let again $\kappa = \boldsymbol{\kappa} \to *$ and $\boldsymbol{\kappa}' = \boldsymbol{\kappa} \to \boldsymbol{\kappa}$ and define for $F : \kappa$, $\boldsymbol{H} : \boldsymbol{\kappa}'$ and $\boldsymbol{Y} : \boldsymbol{\kappa}$ the type $(\mathsf{Lan}_{\boldsymbol{H}}\, F)\boldsymbol{Y}$ by iteration on $|\boldsymbol{\kappa}|$:

$$\begin{aligned} \mathsf{Lan}\, F &:= F \\ (\mathsf{Lan}_{H,\boldsymbol{H}}\, F)\, Y\, \boldsymbol{Y} &:= \exists X^{\kappa_1}.\ HX \leq Y \ \times\ (\mathsf{Lan}_{\boldsymbol{H}}\, (FX))\, \boldsymbol{Y} \end{aligned}$$

Proposition 1. *Let* $F, G : \boldsymbol{\kappa} \to *$ *and* $\boldsymbol{H} : \boldsymbol{\kappa} \to \boldsymbol{\kappa}$. *The following pairs of types are logically equivalent:*

1. $F \leq^{\boldsymbol{H}} G$ *and* $\forall \boldsymbol{X}^{\boldsymbol{\kappa}}.\ F\boldsymbol{X} \to (\mathsf{Ran}_{\boldsymbol{H}}\, G)\boldsymbol{X}$.
2. $F \overset{\boldsymbol{H}}{\leq} G$ *and* $\forall \boldsymbol{Y}^{\boldsymbol{\kappa}}.\ (\mathsf{Lan}_{\boldsymbol{H}}\, F)\boldsymbol{Y} \to G\, \boldsymbol{Y}$.

Proof. Part 1 requires just a close look at the definition of $\leq^{\boldsymbol{H}}$. Part 2 is only slightly more complicated. □

4.2 Embedding

We can simply define the new constants of MIt^{ω} in F^{ω}. Let $\kappa = \boldsymbol{\kappa} \to *$ and $n := |\boldsymbol{\kappa}|$.

$$\begin{array}{lll} \mu_\kappa & : & (\kappa \to \kappa) \to \boldsymbol{\kappa} \to * \\ \mu_\kappa & := & \lambda F^{\kappa\to\kappa} \lambda \boldsymbol{X}^{\boldsymbol{\kappa}} \forall \boldsymbol{H}^{\boldsymbol{\kappa}\to\boldsymbol{\kappa}} \forall G^{\kappa}. (\forall X^{\kappa}.\, X \leq^{\boldsymbol{H}} G \to FX \leq^{\boldsymbol{H}} G) \to (\mathsf{Ran}_{\boldsymbol{H}} G)\boldsymbol{X} \\ \\ \mathsf{GIt}_\kappa & : & \forall F^{\kappa\to\kappa} \forall \boldsymbol{H}^{\boldsymbol{\kappa}\to\boldsymbol{\kappa}} \forall G^{\kappa}.\ (\forall X^{\kappa}.\, X \leq^{\boldsymbol{H}} G \to F\, X \leq^{\boldsymbol{H}} G) \to \mu_\kappa F \leq^{\boldsymbol{H}} G \\ \mathsf{GIt}_\kappa & := & \lambda s \lambda \boldsymbol{f} \lambda r.\, r\, s\, \boldsymbol{f} \\ \\ \mathsf{in}_\kappa & : & \forall F^{\kappa\to\kappa} \forall \boldsymbol{X}^{\boldsymbol{\kappa}}.\ F\, (\mu_\kappa F)\, \boldsymbol{X} \to \mu_\kappa F\, \boldsymbol{X} \\ \mathsf{in}_\kappa & := & \lambda t \lambda s \lambda \boldsymbol{f}.\, s\, (\mathsf{GIt}_\kappa\, s)\, \boldsymbol{f}\, t \end{array}$$

$$\begin{array}{lll} \nu_\kappa & : & (\kappa \to \kappa) \to \boldsymbol{\kappa} \to * \\ \nu_\kappa & := & \lambda F^{\kappa\to\kappa} \lambda \boldsymbol{Y}^{\boldsymbol{\kappa}} \exists \boldsymbol{H}^{\boldsymbol{\kappa}\to\boldsymbol{\kappa}} \exists G^{\kappa}. (\forall X^{\kappa}.\, G \overset{\boldsymbol{H}}{\leq} X \to G \overset{\boldsymbol{H}}{\leq} FX) \times (\mathsf{Lan}_{\boldsymbol{H}} G)\boldsymbol{Y} \\ \\ \mathsf{GCoit}_\kappa & : & \forall F^{\kappa\to\kappa} \forall \boldsymbol{H}^{\boldsymbol{\kappa}\to\boldsymbol{\kappa}} \forall G^{\kappa}.\ (\forall X^{\kappa}.\, G \overset{\boldsymbol{H}}{\leq} X \to G \overset{\boldsymbol{H}}{\leq} F\, X) \to G \overset{\boldsymbol{H}}{\leq} \nu_\kappa F \\ \mathsf{GCoit}_\kappa & := & \lambda s \lambda \boldsymbol{f} \lambda t.\ \ \mathsf{pack}^{n+1}\, \langle s,\, \mathsf{pack}\langle f_1,\, \ldots,\, \mathsf{pack}\langle f_n,\, t\rangle \ldots\rangle\rangle \\ \\ \mathsf{out}_\kappa & : & \forall F^{\kappa\to\kappa} \forall \boldsymbol{Y}^{\boldsymbol{\kappa}}.\ \nu_\kappa F\, \boldsymbol{Y} \to F\, (\nu_\kappa F)\, \boldsymbol{Y} \\ \mathsf{out}_\kappa & := & \lambda r.\ \mathsf{open}\, (r,\, r_1.\mathsf{open}\, (r_1,\, r_2.\ \ldots \mathsf{open}\, (r_{n-1},\, r_n.\mathsf{open}\, (r_n,\, ft_0. \\ & & \quad \mathsf{open}\, (ft_0.1,\, ft_1.\mathsf{open}\, (ft_1.1,\, ft_2.\ \ldots \mathsf{open}\, (ft_{n-1}.1,\, ft_n. \\ & & \quad ft_0.0\ (\mathsf{GCoit}_\kappa\ ft_0.0)\ ft_1.0\ \ldots ft_n.0\ ft_n.1) \ldots)))) \ldots)) \end{array}$$

Theorem 1 (Simulation). *With the definitions above, the following reductions take place in* F^{ω}*:*

$$\begin{array}{lll} \mathsf{GIt}_\kappa\, s\, \boldsymbol{f}\, (\mathsf{in}_\kappa\, t) & \longrightarrow^+ & s\, (\mathsf{GIt}_\kappa\, s)\, \boldsymbol{f}\, t \\ \mathsf{out}_\kappa\, (\mathsf{GCoit}_\kappa\, s\, \boldsymbol{f}\, t) & \longrightarrow^+ & s\, (\mathsf{GCoit}_\kappa\, s)\, \boldsymbol{f}\, t \end{array}$$

Proof. By easy computation.

Corollary 1 (Strong Normalization). *System* MIt^ω *is strongly normalizing, i. e., there is no infinite reduction sequence* $r_0 \longrightarrow r_1 \longrightarrow r_2 \longrightarrow \ldots$ *for any typable term* r_0.

Proof. Use strong normalization of F^ω and simulation.

Since there are no critical pairs in MIt^ω, reduction is locally confluent; by strong normalization and Newman's Lemma, it is confluent on well-typed terms.

5 Conclusion and Related Work

We have proposed MIt^ω, a system of generalized (co)iteration for arbitrary ranks, which turned out to be a definitional extension of Girard's system F^ω and hence enjoys its good meta-theoretic properties, most notably strong normalization. It combines the ideas of Mendler for (co)inductive types with the notion of generalized folds for inductive constructors of rank 2 invented by Bird and Paterson.

MIt^ω has been carefully set up to come with a perspicuous computational behavior, which is very close to general recursion à la `letrec`—a distinctive feature of Mendler-style recursion schemes. For higher ranks, i.e., for the treatment of fixed-points which are themselves type transformers, we described a modified containment relation (via the index $\boldsymbol{H}$) in order to encompass generalized folds, proposed by Bird and Paterson as a means of more elegant definitions of functions operating on nested datatypes.

Therefore, MIt^ω might serve as a basis of a total programming language for nested datatypes. Alternatively, it can be seen as a discipline of programming in existing languages like Haskell which gives termination guarantees for free.

Some related work. The generalized iteration scheme of the present article is a reformulation working in all finite ranks of generalized folds in the liberal sense of Sec. 4.1 and 6 of [8]. More exactly, it extends the "efficient" [11,15] version of that scheme. The efficient generalized folds differ from the original generalized folds in the target type which is constructed with $\leq^{\boldsymbol{H}}$ rather than $\subseteq^{\boldsymbol{H}}$. Here, $\subseteq^{\boldsymbol{H}}$ is the appropriate relativized version of $\subseteq$, defined by

$$F \subseteq^{\boldsymbol{H}} G := \forall \boldsymbol{X}^{\boldsymbol{\kappa}}.F(\boldsymbol{H}\boldsymbol{X}) \to G\boldsymbol{X}.$$

Future work. The realm of higher-rank datatypes seems hardly explored. We certainly wish to try out our schemes on the examples of inductive constructors of rank 3 from [12]. Further, we seek to extend MIt^ω to cover other recursion schemes on nested datatypes like a form of "generalized primitive recursion". This scheme, and others, would no longer have an operationally faithful embedding into System F^ω.

References

1. A. Abel. Termination checking with types. Technical Report 0201, Inst. für Informatik, Ludwigs-Maximilians-Univ. München, 2002.

2. A. Abel and R. Matthes. (Co-)iteration for higher-order nested datatypes. To appear in H. Geuvers, F. Wiedijk, eds., *Post-Conf. Proc. of IST WG TYPES 2nd Ann. Meeting, TYPES'02*, *Lect. Notes in Comput. Sci.*, Springer-Verlag.
3. T. Altenkirch and C. McBride. Generic programming within dependently typed programming. To appear in J. Gibbons and J. Jeuring, *Proc. of IFIP TC2 WC on Generic Programming, WCGP 2002*, Kluwer Acad. Publishers.
4. T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In J. Flum and M. Rodríguez-Artalejo, eds., *Proc. of 13th Int. Wksh. on Computer Science Logic, CSL'99*, vol. 1683 of *Lect. Notes in Comput. Sci.*, pp.53–468. Springer-Verlag, 1999.
5. G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Math. Struct. in Comput. Sci.*, to appear.
6. R. Bird, J. Gibbons, and G. Jones. Program optimisation, naturally. In J. Davies, B. Roscoe, J. Woodcock, eds., *Millenial Perspectives in Computer Science*. Palgrave, 2000.
7. R. Bird and L. Meertens. Nested datatypes. In J. Jeuring, ed., *Proc. of 4th Int. Conf. on Mathematics of Program Construction, MPC'98*, vol. 1422 of *Lect. Notes in Comput. Sci.*, pp. 52–67. Springer-Verlag, 1998.
8. R. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Aspects of Comput.*, 11(2):200–222, 1999.
9. R. Bird and R. Paterson. De Bruijn notation as a nested datatype. *J. of Funct. Program.*, 9(1):77–91, 1999.
10. E. Giménez. Structural recursive definitions in type theory. In *Proc. of 25th Int. Coll. on Automata, Languages and Programming, ICALP'98*, vol. 1443 of *Lect. Notes in Comput. Sci.*, pp. 397–408. Springer-Verlag, 1998.
11. R. Hinze. Efficient generalized folds. In J. Jeuring, ed., *Proc. of 2nd Wksh. on Generic Programming, WGP 2000*, Tech. Report UU-CS-2000-19, Dept. of Comput. Sci., Utrecht Univ., pp. 1–16. 2000.
12. R. Hinze. Manufacturing datatypes. *J. of Funct. Program.*, 11(5): 493–524, 2001.
13. R. Hinze. Polytypic values possess polykinded types. *Sci. of Comput. Program.*, 43(2–3):129–159, 2002.
14. C. B. Jay. Distinguishing data structures and functions: The constructor calculus and functorial types. In S. Abramsky, ed., *Proc. of 5th Int. Conf. on Typed Lambda Calculi and Appl., TLCA'01*, vol. 2044 of *Lect. Notes in Comput. Sci.*, pp. 217–239. Berlin, 2001.
15. C. Martin, J. Gibbons and I. Bayley. Disciplined, efficient, generalised folds for nested datatypes. Submitted.
16. R. Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. PhD thesis, Ludwig-Maximilians-Univ. München, 1998.
17. R. Matthes. Monotone inductive and coinductive constructors of rank 2. In L. Fribourg, ed., *Proc. of 15th Int. Wksh. on Computer Science Logic, CSL 2001*, vol. 2142 of *Lect. Notes in Comput. Sci.*, pp. 600–614. Springer-Verlag, 2001.
18. N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Proc. of 2nd Ann. IEEE Symp. on Logic in Computer Science, LICS'87*, pp. 30–36. IEEE CS Press, 1987.
19. N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Ann. of Pure and Appl. Logic*, 51(1–2):159–172, 1991.
20. S. Mac Lane. *Categories for the Working Mathematician*, vol. 5 of *Graduate Texts in Mathematics*, 2nd ed. Springer-Verlag, 1998.

21. T. Uustalu and V. Vene. A cube of proof systems for the intuitionistic predicate μ-, ν-logic. In M. Haveraaen and O. Owe, eds., *Selected Papers from the 8th Nordic Wksh. on Programming Theory, NWPT '96*, Res. Rep. 248, Dept. of Informatics, Univ. of Oslo, pp. 237–246, 1997.
22. T. Uustalu and V. Vene. Coding recursion à la Mendler (extended abstract). In J. Jeuring, ed., *Proc. of 2nd Wksh. on Generic Programming, WGP 2000*, Tech. Rep. UU-CS-2000-19, Dept. of Comput. Sci., Utrecht Univ., pp. 69–85. 2000.
23. T. Uustalu and V. Vene. The dual of substitution is redecoration. In K. Hammond and S. Curtis, eds., *Trends in Funct. Programming 3*, pp. 99–110. Intellect, 2002.

A System F^ω

In the following we present Curry-style system F^ω enriched with binary sums and products, unit type and existential quantification over constructors. Although we choose a human-friendly notation of variables, we actually mean the nameless version à la de Bruijn which identifies α-equivalent terms. (Capture-avoiding) Substitution of an expression e for a variable x in expression f is denoted by $f[x := e]$.

Kinds are generated from the kind $*$ for types by the binary function kind constructor $\to$:

$$\kappa ::= * \mid \kappa \to \kappa'$$

Constructors. (Denoted by uppercase letters.) Metavariable X ranges over an infinite set of constructor variables.

$$\begin{aligned} A, B, C, F, G ::= {} & X \mid \lambda X^\kappa.F \mid F\,G \mid \forall F^\kappa.\,A \mid \exists F^\kappa.\,A \mid A \to B \\ & \mid A + B \mid A \times B \mid 1 \end{aligned}$$

Equivalence on constructors. Equivalence $F = F'$ for constructors F and F' is given as the compatible closure of the following axiom.

$$(\lambda X.F)\,G =_\beta F[X := G]$$

We identify constructors up to equivalence, which is a decidable relation due to normalization and confluence of simply-typed λ-calculus (where our constructors are the terms and our kinds are the types of that calculus).

Objects (Terms). (Denoted by lowercase letters) The metavariable x ranges over an infinite set of object variables.

$$\begin{aligned} r, s, t ::= {} & x \mid \lambda x.t \mid r\,s \mid \mathsf{inl}\,t \mid \mathsf{inr}\,t \mid \mathsf{case}\,(r,\, x.\,s,\, y.\,t) \\ & \mid \langle\rangle \mid \langle t_0, t_1\rangle \mid r.0 \mid r.1 \mid \mathsf{pack}\,t \mid \mathsf{open}\,(r,\, x.\,s) \end{aligned}$$

Contexts. Variables in a context Γ are assumed to be distinct.

$$\Gamma ::= \cdot \mid \Gamma, X^\kappa \mid \Gamma, x : A$$

Judgments. (Simultaneously defined)

$\Gamma\ \mathsf{cxt}$	Γ is a wellformed context
$\Gamma \vdash F : \kappa$	F is a wellformed constructor of kind κ in context Γ
$\Gamma \vdash t : A$	t is a wellformed term of type A in context Γ

Wellformed contexts. $\Gamma\ \mathsf{cxt}$

$$\frac{}{\cdot\ \mathsf{cxt}} \qquad \frac{\Gamma\ \mathsf{cxt}}{\Gamma, X^\kappa\ \mathsf{cxt}} \qquad \frac{\Gamma \vdash A : *}{\Gamma, x{:}A\ \mathsf{cxt}}$$

Wellkinded constructors. $\Gamma \vdash F : \kappa$

$$\frac{X^\kappa \in \Gamma \qquad \Gamma\ \mathsf{cxt}}{\Gamma \vdash X : \kappa} \qquad \frac{\Gamma, X^\kappa \vdash F : \kappa'}{\Gamma \vdash \lambda X^\kappa.F : \kappa \to \kappa'} \qquad \frac{\Gamma \vdash F : \kappa \to \kappa' \qquad \Gamma \vdash G : \kappa}{\Gamma \vdash F\,G : \kappa'}$$

$$\frac{\Gamma, X^\kappa \vdash A : *}{\Gamma \vdash \forall X^\kappa.\, A : *} \qquad \frac{\Gamma, X^\kappa \vdash A : *}{\Gamma \vdash \exists X^\kappa.\, A : *} \qquad \frac{\Gamma \vdash A : * \qquad \Gamma \vdash B : *}{\Gamma \vdash A \to B : *}$$

$$\frac{\Gamma \vdash A : * \qquad \Gamma \vdash B : *}{\Gamma \vdash A + B : *} \qquad \frac{\Gamma \vdash A : * \qquad \Gamma \vdash B : *}{\Gamma \vdash A \times B : *} \qquad \frac{\Gamma\ \mathsf{cxt}}{\Gamma \vdash 1 : *}$$

Welltyped terms. $\Gamma \vdash t : A$

$$\frac{(x{:}A) \in \Gamma \qquad \Gamma\ \mathsf{cxt}}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\,s : B}$$

$$\frac{\Gamma, X^\kappa \vdash t : A}{\Gamma \vdash t : \forall X^\kappa.A} \qquad \frac{\Gamma \vdash t : \forall X^\kappa.A \qquad \Gamma \vdash F : \kappa}{\Gamma \vdash t : A[X := F]}$$

$$\frac{\Gamma \vdash t : A[X := F] \qquad \Gamma \vdash F : \kappa}{\Gamma \vdash \mathsf{pack}\,t : \exists X^\kappa.A} \qquad \frac{\Gamma \vdash r : \exists X^\kappa.A \qquad \Gamma, X^\kappa, x{:}A \vdash s : C}{\Gamma \vdash \mathsf{open}\,(r,\, x.\, s) : C}$$

$$\frac{\Gamma\ \mathsf{cxt}}{\Gamma \vdash \langle\rangle : 1} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : *}{\Gamma \vdash \mathsf{inl}\,t : A + B} \qquad \frac{\Gamma \vdash t : B \qquad \Gamma \vdash A : *}{\Gamma \vdash \mathsf{inr}\,t : A + B}$$

$$\frac{\Gamma \vdash r : A + B \qquad \Gamma, x{:}A \vdash s : C \qquad \Gamma, y{:}B \vdash t : C}{\Gamma \vdash \mathsf{case}\,(r,\, x.\, s,\, y.\, t) : C}$$

$$\frac{\Gamma \vdash t_0 : A_0 \qquad \Gamma \vdash t_1 : A_1}{\Gamma \vdash \langle t_0, t_1\rangle : A_0 \times A_1} \qquad \frac{\Gamma \vdash r : A_0 \times A_1 \quad i \in \{0,1\}}{\Gamma \vdash r.i : A_i}$$

Reduction. The one-step reduction relation $t \longrightarrow t'$ between terms t and t' is defined as the closure of the following axioms under all term constructors.

$$\begin{array}{lll}
(\lambda x.t)\, s & \longrightarrow_\beta t[x := s] & \\
\mathsf{case}\,(\mathsf{inl}\, r,\ x.\, s,\ y.\, t) & \longrightarrow_\beta s[x := r] & \\
\mathsf{case}\,(\mathsf{inr}\, r,\ x.\, s,\ y.\, t) & \longrightarrow_\beta t[y := r] & \\
\langle t_0, t_1 \rangle.i & \longrightarrow_\beta t_i & \text{if } i \in \{0, 1\} \\
\mathsf{open}\,(\mathsf{pack}\, t,\ x.\, s) & \longrightarrow_\beta s[x := t] &
\end{array}$$

We denote the transitive closure of $\longrightarrow$ by $\longrightarrow^+$ and the reflexive-transitive closure by $\longrightarrow^*$.

The defined system is a conservative extension of system F^ω. Reduction is type-preserving, confluent and strongly normalizing.

Ambiguous Classes in the Games μ-Calculus Hierarchy

André Arnold and Luigi Santocanale*

LaBRI - Université Bordeaux 1
{andre.arnold,santocan}@labri.fr

Abstract. Every parity game is a combinatorial representation of a closed Boolean μ-term. When interpreted in a distributive lattice every Boolean μ-term is equivalent to a fixed-point free term. The alternation-depth hierarchy is therefore trivial in this case. This is not the case for non distributive lattices, as the second author has shown that the alternation-depth hierarchy is infinite.
In this paper we show that the alternation-depth hierarchy of the games μ-calculus, with its interpretation in the class of all complete lattices, has a nice characterization of ambiguous classes: every parity game which is equivalent both to a game in Σ_{n+1} and to a game in Π_{n+1} is also equivalent to a game obtained by composing games in Σ_n and Π_n.

Introduction

Parity games have shown to be a fundamental tool in the theory of automata recognizing infinite objects and of the logics by which these objects are usually defined [23]. Among these logics we list monadic second order logic, the propositional modal μ-calculus, and the collection of their fragments, i.e. logics of computation such as PDL, LTL, CTL, etc. The use of the games is not restricted to the theory but carries over to applications such as model-checking [9] or the synthesis of controllers [4].

In the monograph [3] parity games are used to establish strong relationships between μ-calculi and classes of automata (on words, on trees, on Kripke structures, etc.) A class of automata is given the structure of a μ-calculus by defining a composition operation $\mathcal{A}[\mathcal{B}/x]$ on automata and two fixed-point operations $\mu x.\mathcal{A}$ and $\nu x.\mathcal{A}$. Recall that a μ-calculus is a set of syntactical entities with an intended functional interpretation on a complete lattice L, each term t of arity $ar(t)$ being interpreted as a monotonic mapping from $L^{ar(t)}$ to L. The terms $\theta x.t$ of a μ-calculus, for $\theta \in \{\mu, \nu\}$, are interpreted as extremal fixed-points of t, so that $\theta x.t$ and $t[\theta x.t/x]$ denote the same object. In the μ-calculus of automata, the complete lattice L is a powerset of a set of objects (words, trees, etc.); the interpretation of an automaton, as a term of empty arity of the μ-calculus, coincides with the language of objects it accepts.

* The second author acknowledges financial support from the European Commission through an individual Marie Curie fellowship.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 70–86, 2003.

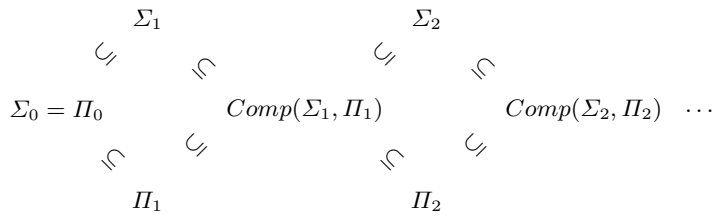


Fig. 1. The alternation-depth hierarchy

Parity games are syntactically similar to automata, thus the composition and fixed-point operations can be defined on games as well. In the monograph [3] this was not done, as, by analogy with the case of automata, the interpretation of a game is whether some distinguished position is winning or not. Since there are only two objects in the interpretation domain, such a μ-calculus of games is not very interesting.[1] On the other hand, for a given μ-calculus, it is also possible to consider as the intended interpretation a given class of complete lattices. It turns out that considering the class of distributive (complete) lattices is not enough to make the μ-calculus of games nontrivial. In this case, every term is equivalent to a term with no applications of the fixed-point operations. In order to have a μ-calculus of games with a nontrivial interpretation one needs to consider the class of all complete lattices. In [20] a μ-calculus with such interpretation is considered. It is defined a preorder $\leq$ on the collection $\mathcal{G}$ of games, in a constructive way. The quotient of this collection of games under the equivalence relation $\sim$ on $\mathcal{G}$ induced by $\leq$ is a lattice (although not a complete one) where the interpretation of $\theta x.G$ is indeed an extremal fixed-point. This is moreover the universal μ-lattice, that is the universal lattice in which every μ-term has an interpretation. By saying that this algebraic object is universal we mean that two terms s, t satisfy $s \leq t$ in the quotient $\mathcal{G}/\sim$ if and only if this relation holds in every lattice where all μ-terms are interpretable. Moreover the relation $s \leq t$ holds in $\mathcal{G}/\sim$ if and only if it holds in every complete lattice.

The extremal fixed-point operations of μ-calculi are syntactic operators analogous to quantifiers. There have been few proposals to classify μ-terms into classes according to the number of nested applications of fixed-point operations; most of these classifications happen to be equivalent [14, 15, 16]. We recall here the hierarchy of μ-terms into classes proposed in [15]. The class $\Sigma_0 = \Pi_0$ is the class of μ-terms with no application of the fixed-point operations; Σ_{n+1} (resp. Π_{n+1}) is the closure of Σ_n and Π_n under the composition operation and the least fixed-point operation (resp. the greatest fixed-point operation). Also, the class $Comp(\Sigma_n, \Pi_n)$ is defined as the closure of Σ_n and Π_n under the composition operation. These classes are ordered by the inclusions as shown in figure 1. As far as we are dealing with the syntax, these inclusions are obviously strict. However,

[1] It is also possible to say that the interpretation of a game is the set of winning strategies for a given player. This idea was pursued in [21].

if a μ-calculus comes with an intended interpretation, the relevant question is whether these inclusions are strict in the interpretation, and this question has no obvious answer. For the propositional modal μ-calculus the semantical strictness of these inclusions was proved in [6, 13, 1]. For the μ-calculus of parity games with its interpretation in the class of all complete lattices these inclusions were shown to be strict in [18].

In this paper we investigate an orthogonal problem. It is easily seen that the relation

$$Comp(\Sigma_n, \Pi_n) = \Sigma_{n+1} \cap \Pi_{n+1} \tag{1}$$

holds in every μ-calculus, at least at the syntactic level; it can be asked whether such equality still holds with respect to a given interpretation. This question is inspired by the characterization of the ambiguous classes in the Borel hierarchy [12, §11.B,§22.27], of which, there are already known analogies within μ-calculi. Indeed, it is an easy exercise to show that if a language of infinite words is accepted both by a deterministic automaton in Σ_{n+1} and by a deterministic automaton in Π_{n+1}, then this language is accepted by a deterministic automaton in $Comp(\Sigma_n, \Pi_n)$. We remark that questions related to the alternation-depth hierarchy for deterministic automata on infinite words tend to be easy, for example this hierarchy is even decidable [17]. Also, it was shown in [2] that if language of trees is definable both by an automaton in Σ_2 and by an automaton in Π_2, then it is also definable by an automaton in $Comp(\Sigma_1, \Pi_1)$.

We show that, for the μ-calculus of games with its interpretation in the class of all complete lattices, the equality (1) holds semantically, for every $n \geq 0$. As far as we know, together with the observation that this equality holds for languages of infinite words defined by deterministic automata, this is the first complete result of this type for μ-calculi. We do not answer the analogous question for other μ-calculi but we believe that the ideas and tools presented here can be adapted to other contexts. These tools are proof-theoretic in nature: the main technical proposition is an interpolation theorem[2] that we prove essentially with Maehara's method [22, §1.6.5]. The proofs that we consider here are however circular, see [19], and being able to apply existing proof-theoretic techniques is not straightforward.

The paper is organized as follows. In section 1 we define parity games with draws and the operations on these games. We also define their semantics as monotonic mappings on a complete lattice. In section 2 we organize parity games with draws into a μ-calculus and define the syntactical preorder that characterizes the semantical order relation. In section 3, we firstly recall the definition and basic facts about the hierarchy; then we prove that equality (1) holds at the semantical level.

[2] This interpolation result concerns the hierarchy of fixed points; it contrasts with the uniform interpolation property of the modal μ-calculus [7] which concerns the common language of two formulas and does not take into account the hierarchy of fixed points, since the main tool to prove it are disjunctive normal forms.

1 Parity Games with Draws

A *parity game with draws* is a tuple $G = \langle Pos_E^G, Pos_A^G, Pos_D^G, M^G, \rho^G \rangle$ where:

- $Pos_E^G, Pos_A^G, Pos_D^G$ are finite pairwise disjoint sets of positions (Eva's positions, Adam's positions, and draw positions),
- M^G, the set of moves, is a subset of $(Pos_E^G \cup Pos_A^G) \times (Pos_E^G \cup Pos_A^G \cup Pos_D^G)$,
- ρ^G is a mapping from $(Pos_E^G \cup Pos_A^G)$ to $\mathbb{N}$.

Whenever an initial position is specified, these data define a game between player Eva and player Adam. The outcome of a finite play is determined according to the normal play condition: a player who cannot move loses. It can also be a draw, if a position in Pos_D^G is reached.[3] The outcome of an infinite play $\{ (g_k, g_{k+1}) \in M^G \}_{k \geq 0}$ is determined by means of the rank function ρ^G as follows: it is a win for Eva if and only if the number

$$\max \{ i \in \mathbb{N} \mid \exists \text{ infinitely many } k \text{ s.t. } \rho^G(g_k) = i \}$$

is even. To simplify the notation, we shall use $Pos_{E,A}^G$ for the set $Pos_E^G \cup Pos_A^G$ and use similar notations such as $Pos_{E,D}^G$, etc. We let $Max^G = \max \rho^G(Pos_{E,A}^G)$ if the set $Pos_{E,A}^G$ is not empty, and $Max^G = -1$ otherwise.

1.1 Operations on Parity Games

We define here some operations and constants on games. When defining operations on games we shall always assume that the sets of positions of distinct games are pairwise disjoint.

Meets and Joins. For any finite set I, $\bigwedge_I$ is the game defined by letting $Pos_E = \emptyset$, $Pos_A = \{p_0\}$, $Pos_D = I$, $M = \{ (p_0, i) \mid i \in I \}$ (where $p_0 \notin I$), $\rho(p_0) = 0$. The game $\bigvee_I$ is defined similarly, exchanging Pos_E and Pos_A.

Composition Operation. Given two games G and H and a mapping $\psi : P_D^G \longrightarrow P_{E,A,D}^H$, the game $K = G \circ_\psi H$ is defined as follows:

- $Pos_E^K = Pos_E^G \cup Pos_E^H$,
- $Pos_A^K = Pos_A^G \cup Pos_A^H$,
- $Pos_D^K = Pos_D^H$,
- $M^K = (M^G \cap (Pos_{E,A}^G \times Pos_{E,A}^G)) \cup M^H$
  $\cup \{ (p, \psi(p')) \mid (p, p') \in M^G \cap (Pos_{E,A}^G \times Pos_D^G) \}$.
- ρ^K is such that its restrictions to the positions of G and H are respectively equal to ρ^G and ρ^H.

Sum Operation. Given a finite collection of parity games G_i, $i \in I$, their sum $H = \sum_{i \in I} G_i$ is defined in the obvious way:

- $P_Z^H = \bigcup_{i \in I} P_Z^{G_i}$, for $Z \in \{E, A, D\}$,
- $M^H = \bigcup_{i \in I} M^{G_i}$,
- ρ^H is such that its restriction to the positions of each G_i is equal to ρ_i^G.

[3] Observe that there are no possible moves from a position in Pos_D^G.

Fixed-point Operations. If G is a game, a system on G is a tuple $S = \langle E, A, M\rangle$ where:

- E and A are pairwise disjoint subsets of Pos_D^G,
- $M \subseteq (E \cup A) \times Pos_{E,A,D}^G$.

Given a system S and $\theta \in \{\mu, \nu\}$, we define the parity game $\theta_S.G$:

- $Pos_E^{\theta_S.G} = Pos_E^G \cup E$,
- $Pos_A^{\theta_S.G} = Pos_A^G \cup A$,
- $Pos_D^{\theta_S.G} = Pos_D^G - (E \cup A)$,
- $M^{\theta_S.G} = M^G \cup M$,
- $\rho^{\theta_S.G}$ is the extension of ρ^G to $E \cup A$ such that:
  - if $\theta = \mu$, then $\rho^{\theta_S.G}$ takes on $E \cup A$ the constant value Max^G if this number is odd or $Max^G + 1$ if Max^G is even,
  - if $\theta = \nu$, then $\rho^{\theta_S.G}$ takes on $E \cup A$ the constant value Max^G if this number is even or $Max^G + 1$ if Max^G is odd.

Predecessor Game. Let G be a game such that $Max^G \neq -1$, i.e. there is at least one position in $Pos_{E,A}^G$. Let $Top^G = \{\, g \in Pos_{E,A}^G \mid \rho^G(g) = Max^G \,\}$, then the predecessor game G^- is defined as follows:

- $Pos_E^{G^-} = Pos_E^G - Top^G$,
- $Pos_A^{G^-} = Pos_A^G - Top^G$,
- $Pos_D^{G^-} = Pos_D^G \cup Top^G$,
- $M^{G^-} = M^G - (Top^G \times Pos_{E,A,D}^G)$,
- ρ^{G^-} is the restriction of ρ^G to $Pos_{E,A}^{G^-}$.

1.2 Semantics of Parity Games

Given a complete lattice L, the intepretation of a parity game G is going to be a monotone mapping of the form

$$\|G\| : L^{P_D^G} \longrightarrow L^{P_{E,A}^G},$$

where $L^{P_K^G}$ is the P_K^G-fold product lattice of L with itself. If $g \in Pos_{A,E}^G$ then $\|G_g\|$ will denote the projection of $\|G\|$ onto the g coordinate. Any parity game G can be reconstructed in a unique way from the predecessor game G^- by one application of some fixed-point operation θ_S; moreover the predecessor game is "simpler". Thus we define the interpretation of a parity game inductively. If $P_{E,A}^G = \emptyset$, then $L^{P_{E,A}^G} = L^\emptyset = 1$, the complete lattices with just one element, and there is just one possible definition of the mapping $\|G\|$. Otherwise, if Max^G is odd, then $\|G\|$ is the parameterized least fixed-point of the monotone mapping

$$L^{P_{E,A}^G} \times L^{P_D^G} \longrightarrow L^{P_{E,A}^G}$$

defined by the system of equations:

$$x_g = \begin{cases} \bigvee \{ x_{g'} \mid (g,g') \in M^G \} & \text{if } g \in Pos_E^G \cap Top^G, \\ \bigwedge \{ x_{g'} \mid (g,g') \in M^G \} & \text{if } g \in Pos_A^G \cap Top^G, \\ \|G_g^-\|(X_{Top^G}, X_{Pos_D^G}) & \text{otherwise.} \end{cases}$$

If Max^G is even, then $\|G\|$ is the parameterized greatest fixed-point of this mapping.

2 The μ-Calculus of Games

Let X be a countable set of variables. A pointed parity game with labeled draws is a tuple $\langle G, p_\star^G, \lambda^G \rangle$ where G is a parity game, $p_\star^G \in Pos_{E,A,D}^G$ is a specified initial position, and $\lambda^G : Pos_D^G \longrightarrow X$ is a labeling of draw positions by variables. With $\mathcal{G}$ we shall denote the collection of all pointed parity games with labeled draws; as no confusion will arise, we will call a pointed parity game with labeled draws simply "game". Similarly, we shall abuse the notation and write G to denote the entire tuple $\langle G, p_\star^G, \lambda^G \rangle$. With the notation G_g we shall denote the game that differs from G only in that the initial position is now g, i.e $p_\star^{G_g} = g$.

We give the collection $\mathcal{G}$ the structure of a μ-calculus, as defined in [3, §2.1]. If x is a variable, the game $\hat{x}$ has just one draw position labeled by x. The arity of a game G, denoted by $ar(G)$, is the subset of variables $\lambda^G(Pos_D^G)$.

A substitution is a mapping $\sigma : X \longrightarrow \mathcal{G}$; given a game G and a substitution σ, the composition of G and σ – for which we use the notation $G[\sigma]$ – is defined as

$$G[\sigma] = (G \circ_\psi \sum_{x \in ar(G)} \sigma(x)) \,,$$

where $\psi(g) = p_\star^{\sigma(\lambda^G(g))}$ for $g \in Pos_D^G$. Moreover, $p_\star^{G[\sigma]} = p_\star^G$ and $\lambda^{G[\sigma]}(p) = \lambda^{\sigma(x)}(p)$ whenever $p \in Pos_D^{\sigma(x)}$. Therefore $ar(G[\sigma]) = \bigcup_{x \in ar(G)} ar(\sigma(x))$.

Similarly, given G in $\mathcal{G}$ and $x \in X$, let $Pos_x = \{ g \in Pos_D^G \mid \lambda^G(g) = x \}$. Define the system S as $\langle \emptyset, Pos_x, Pos_x \times \{p_\star^G\} \rangle$. Then we define

$$\theta x.G = (\theta_S.G) \,,$$

where moreover $\lambda^{\theta x.G}$ is the restriction of λ^G and $p_\star^{\theta x.G} = p_\star^G$. Remark that $\theta x.G = G$ if $x \notin ar(G)$.

The above constructions are analogous to those given in [3, §7.2] for automata and therefore one can mimic the proof presented there to show that $\mathcal{G}$ with this structure satisfies the axioms of a μ-calculus.

Observe that the operation of forming the predecessor game G^- can be extended to pointed parity games with labeled draws if we choose a variable $x_g \notin ar(G)$ for each $g \in Top^G$: we let in this case λ^{G^-} be the extension of λ^G such that $\lambda^{G^-}(g) = x_g$ for $g \in Top^G$.

2.1 The Preorder on $\mathcal{G}$

In order to describe a preorder on the class $\mathcal{G}$, we shall define a new game $\langle\langle G, H\rangle\rangle$ for a pair of games G and H in $\mathcal{G}$. This is not a pointed parity game with draws as defined in the previous section; to emphasize this fact, the two players will be named Mediator and Opponent instead of Eva and Adam.

Before formally defining the game $\langle\langle G, H\rangle\rangle$, we give its informal description and explanation. Mediator's goal is to prove that the relation $\|G\| \leq \|H\|$ holds in any complete lattice; Opponent's goal is to show that this relation does not hold. For example, if $G = \bigvee_{i\in I} G_i$ has the shape of a join and $H = \bigwedge_{j\in J} H_j$ has the shape of a meet, then this is an Opponent's position: Mediator should be prepared to prove $\|G_i\| \leq \|H_j\|$ for any pair of indexes i and j; thus Opponent should find a pair of indexes (i, j) and show that $\|G_i\| \not\leq \|H_j\|$. If $G = \bigwedge_{i\in I} G_i$ is a meet and $H = \bigvee_{j\in J} H_j$ is a join, then this is a Mediator's position: Mediator should find either an i and show that $\|G_i\| \leq \|H\|$ or a j and show that $\|G\| \leq \|H_j\|$; Opponent should be prepared to disprove any such relation.[4]

Thus the game is played on the two boards, simultaneously. At a first approximation, a position of $\langle\langle G, H\rangle\rangle$ is a pair of positions from G and H. Since we code meets as Adam's positions and joins as Eva's positions, Mediator is playing with Adam on G and with Eva on H; Opponent is playing with Eva on G and with Adam on H. Thus a pair (g, h) in $Pos^G_A \times Pos^H_E$ clearly belongs to Mediator and a pair (g, h) in $Pos^G_E \times Pos^H_A$ clearly belongs to Opponent. Pairs in $Pos^G_E \times Pos^H_E$ or $Pos^G_A \times Pos^H_A$ are ambiguous, as both players could play. The situation is not symmetric, however, as Opponent is obliged to play while Mediator is allowed to play, if he wants, but he can also decide to delay his move. In the formal definition, we code the fact that two players can play from the same pair by duplicating every pair into a Mediator's position and into an Opponent's position.

Definition 2.1. The game $\langle\langle G, H\rangle\rangle$ is defined as follows:

- The set of Mediator's positions is

$$Pos^G_{E,A,D} \times \{M\} \times Pos^H_{E,A,D}\,,$$

  and the set of Opponent's positions is

$$Pos^G_{E,A,D} \times \{O\} \times Pos^H_{E,A,D}\,.$$

- We describe the moves[5] by cases:
  - If $(g, h) \in (Pos^G_E \times Pos^H_{A,D}) \cup (Pos^G_{E,D} \times Pos^H_A)$, then there is just one "silent" move

$$(g, M, h) \to (g, O, h)$$

[4] These moves suffice to Mediator to reach his goal, as the relation $\leq$ that we shall define turns out to be transitive. This fact is analogous to a cut-elimination theorem and to Whitman's conditions characterizing free lattices [10].

[5] As we wish to distinguish moves coming from G and moves coming from H, the underlying graph of this game can have distinct edges relating the same pair of vertices.

and moves of the form

$$(g, O, h) \to (g', M, h) \qquad (g, O, h) \to (g, M, h')$$

for every move $(g, g') \in M^G$ and every move $(h, h') \in M^H$.

- If $(g, h) \in (Pos^G_A \times Pos^H_{E,D}) \cup (Pos^G_{A,D} \times Pos^H_E)$, then there is just one silent move

$$(g, O, h) \to (g, M, h)$$

and moves of the form

$$(g, M, h) \to (g', O, h) \qquad (g, M, h) \to (g, O, h')$$

for every move $(g, g') \in M^G$ and every move $(h, h') \in M^H$.

- If $(g, h) \in (Pos^G_E \times Pos^H_E)$ then there are moves of the form

$$(g, O, h) \to (g', M, h) \qquad (g, M, h) \to (g, O, h')$$

for every move $(g, g') \in M^G$ and every move $(h, h') \in M^H$, and moreover a silent move

$$(g, M, h) \to (g, O, h)\,.$$

- Similarly, if $(g, h) \in (Pos^G_A \times Pos^H_A)$ then there are moves of the form

$$(g, M, h) \to (g', O, h) \qquad (g, O, h) \to (g, M, h')$$

for every move $(g, g') \in M^G$ and every move $(h, h') \in M^H$, and moreover a silent move

$$(g, M, h) \to (g, O, h)\,.$$

- Finally, if $(g, h) \in Pos^G_D \times Pos^H_D$, then: If $\lambda^G(g) = \lambda^H(h)$, then there is a move

$$(g, M, h) \to (g, O, h)$$

and no move from (g, O, h). Hence this is a winning position for Mediator. If $\lambda^G(g) \neq \lambda^H(h)$, then there is a move

$$(g, O, h) \to (g, M, h)$$

and no move from (g, M, h). The latter is a win for Opponent.

– Now let us define the winning plays for Mediator in this game. As usual a maximal finite play is lost by the player who cannot move. For infinite plays, observe that any (maximal) play γ in $\langle\langle G, H\rangle\rangle$ defines two plays (not necessarily maximal) $\pi_G(\gamma)$ in G and $\pi_H(\gamma)$ in H. Generalizing what happens for finite plays we say that Mediator wins an infinite play γ if and only if either $\pi_G(\gamma)$ is a win for Adam on G, or $\pi_H(\gamma)$ is a win for Eva on H.

In the above definition we must explain the meaning of statements such as "$\pi_H(\gamma)$ is a win for Eva on H" whenever $\pi_H(\gamma)$ is a finite play which is not maximal. In this case, the last position of the play $\pi_H(\gamma)$ belongs either to Pos_E^H or to Pos_A^H: we say that this is a win for Adam in the first case and a win for Eva in the latter case, with the intuition that the player who gives up playing loses.

This convention allows Mediator to play just on one board and to give up on the other if Adam has a winning strategy on G or Eva has a winning strategy on H. On the other hand, as soon as Opponent gives up on one board, he's going to lose. Notice that the game $\langle\!\langle G, H \rangle\!\rangle$ alternates between Opponent's positions and Mediator's positions, thus if a player among Mediator and Opponent gives up on one board, this is indeed his own responsibility.

Finally observe that the condition (1): "$\pi_G(\gamma)$ is a win for Adam on G, or $\pi_H(\gamma)$ is a win for Eva on H" implies but is not equivalent to (2): "if $\pi_G(\gamma)$ is a win for Eva on G, then $\pi_H(\gamma)$ is a win for Eva on H". The logic is complicated by the fact that $\pi_G(\gamma)$ could be a draw, but this is also the only obstacle to obtain the equivalence between (1) and (2).

Definition 2.2. If G and H belong to $\mathcal{G}$, then we declare that $G \leq H$ if and only if Mediator has a winning strategy in the game $\langle\!\langle G, H \rangle\!\rangle$ starting from position $(p_\star^G, O, p_\star^H)$.

In the following, we shall write $G \sim H$ to mean that $G \leq H$ and $H \leq G$. We continue by listing some useful facts about the game $\langle\!\langle G, H \rangle\!\rangle$ and the relation $\leq$.

Lemma 2.3. *In the game $\langle\!\langle G, H \rangle\!\rangle$ Mediator has a winning strategy from a position of the form (g, O, h) if and only if he has a winning strategy from (g, M, h).*

We do not include a proof of this lemma for lack of space.

An *homomorphism* from a game G to a game H is a mapping f from the positions of G to the positions of H such that:

- $f(p_\star^G) = h_\star^H$,
- if g belongs to Pos_E^G (resp. Pos_A^G) then $f(g)$ belongs to Pos_E^H (resp. Pos_A^H) and $\rho^G(g) = \rho^H(f(g))$,
- if g belongs to Pos_D^G then $f(g)$ belongs to Pos_D^H and $\lambda^G(g) = \lambda^H(f(g))$,
- if $(g, g') \in M^G$ then $(f(g), f(g')) \in M^H$.

An homomorphism f from a game G to a game H is a *bisimulation* if moreover:

- for any position g of G, if $(f(g), h) \in M^H$ then there exists a position g' of G such that $(g, g') \in M^G$ and $h = f(g')$.

Lemma 2.4. *If there is a bisimulation from G to H, then $G \sim H$.*

The proof of this lemma, which we do not include for lack of space, amounts to observe that both in the game $\langle\langle G, H\rangle\rangle$ and the game $\langle\langle H, G\rangle\rangle$ Mediator can use a "copycat" strategy. We can use Lemma 2.4 to establish several equivalences. Let G be a game and $T \subseteq Pos^G_{E,A}$ be a collection of positions of G. Let $X_T \subseteq X$ be a subset of variables in bijection with T and such that $X_T \cap ar(G) = \emptyset$. The game $G^{T \leadsto X_T}$ is obtained as follows: every position $t \in T$ is added to the set of draw positions and labeled by the variable x_t which corresponds to t. Of course there are no more moves from a position $t \in T$ in the game $G^{T \leadsto X_T}$. The relation $G_g \sim G_g^{T \leadsto X_T}[G_t/x_t]_{t \in T}$ holds, as a consequence of the fact that there is a bisimulation from $G_g^{T \leadsto X_T}[G_t/x_t]$ to G_g. Also, let G'_g be the game obtained from G_g by considering the reachable part from g. Again, we have $G_g \sim G'_g$ as the inclusion of the positions of G'_g into the positions of G_g is a bisimulation. Thus we are allowed to consider only games in $\mathcal{G}$ that are reachable from the initial position.

Proposition 2.5. *The relation $\leq$ has the following properties:*

1. *It is reflexive and transitive.*
2. *Composition is monotonic: If $G \leq H$ and if for all $x \in X$, $\sigma(x) \leq \sigma'(x)$ then $G[\sigma] \leq H[\sigma']$.*
3. *For any game G and any substitution σ, $G \leq \bigwedge_I[\sigma]$ if and only if $G \leq \sigma(x_i)$ for all $i \in I$.*
4. *For any game H and any substitution σ, $\bigvee_I[\sigma] \leq H$ if and only if $\sigma(x_i) \leq H$ for all $i \in I$.*
5. *For $\theta \in \{\mu, \nu\}$, $\theta x.G \sim G[\theta x.G/x]$.*
6. *If $G[H/x] \leq H$ then $\mu x.G \leq H$.*
7. *If $G \leq G[H/x]$ then $G \leq \nu x.H$.*
8. *It is the least relation on $\mathcal{G}$ having properties 1 to 7.*
9. *It is sound and complete with respect to the class of all complete lattices: $G \leq H$ if and only if for any complete lattice L and any $v : X \longrightarrow L$*

$$\|G_{p^G_\star}\|(v \circ \lambda^G) \leq \|H_{p^H_\star}\|(v \circ \lambda^H)\,.$$

Proof. These properties were stated and proved in [20] for a restricted class of fair games and for a different relation $\preceq$ (similar to the one of [5, 11]). However, we can prove the following: 1) the relation $\leq$ is indeed reflexive, transitive, and monotonic, 2) every game in $\mathcal{G}$ is $\leq$-equivalent to a fair game, 3) if G and H are fair games, then $G \leq H$ if and only if $G \preceq H$.

Therefore the quotient of the class of fair games under the equivalence relation induced by $\preceq$ is order isomorphic to the quotient of $\mathcal{G}$ under the equivalence relation $\sim$ and this quotient inherits all the properties proved in [20]. □

In particular the quotient $\mathcal{G}/\sim$ is a lattice where the greatest lower bound (resp. least upper bound) of the equivalence classes of $G_1, \ldots, G_k$ is the equivalence class of $\bigwedge_k(G_1, \ldots, G_k)$ (resp. $\bigvee_k(G_1, \ldots, G_k)$). It is a *$\mu$-lattice* as well, meaning that all the μ-terms constructible from the signature $\langle \top, \wedge, \bot, \vee \rangle$ are interpretable as infima, suprema, least prefixed points and greatest postfixed

points of previously defined operations. The μ-lattice $\mathcal{G}/\sim$ is freely generated by the set X, meaning that given any μ-lattice L and any mapping $\psi : X \longrightarrow L$, there exists a unique extension of ψ to a mapping $\psi' : \mathcal{G}/\sim \longrightarrow L$ that preserves the interpretation of μ-terms. From this property it readily follows that $\leq$ is the least preorder having the properties listed above.

3 Ambiguous Classes in the Mostowski Hierarchy

If G is a game then two mappings ρ and ρ' from $Pos^G_{E,A}$ to $\mathbb{N}$ are said to be equivalent with respect to G if any infinite path in G is winning according to ρ if and only if it is winning according to ρ'. Let G be a game and ρ be a mapping equivalent to ρ^G w.r.t. G. It is easily observed that the game G' obtained from G by substituting the rank function ρ with ρ^G is equivalent to G: $G \sim G'$.

Definition 3.1. We say that a game G belongs to $\Sigma_0 = \Pi_0$ if and only if it is acyclic. For $n \geq 1$, we say that a game G belongs to Σ_n (resp. Π_n) if there is a mapping ρ equivalent to ρ^G w.r.t. G, and an odd (resp. even) number $m \geq n-1$ such that $\rho(Pos^G_{E,A}) \subseteq \{m-n+1, \ldots, m\}$. We say that a game belongs to $Comp(\Sigma_n, \Pi_n)$ if it can be obtained from games in Σ_n and Π_n by a sequence of applications of the composition operation.

Observe that, by construction, for every $n \geq 1$, if G belongs to Σ_n (resp. Π_n) then $\mu x.G$ belongs to Σ_n (resp. Π_{n+1}) and $\nu x.G$ belongs to Σ_{n+1} (resp. Π_n). Moreover, $Comp(\Sigma_0, \Pi_0) = \Sigma_0$ and in general $Comp(\Sigma_n, \Pi_n) \subseteq \Sigma_{n+1} \cap \Pi_{n+1}$. We shall show that the converse holds as well.

Lemma 3.2. *If a game G belongs to $\Sigma_1 \cap \Pi_1$ then it is acyclic.*

Proof. As G belongs to $\Sigma_1 \cap \Pi_1$ there are two mappings ρ and ρ' equivalent to ρ^G w.r.t. G whose images are respectively $\{m\}$ and $\{m'\}$, where m is odd and m' is even. If G is not acyclic, there exists a position p in G and a non empty path γ from p to p. The infinite path γ^ω is a win for Adam, according to ρ, and a win for Eva, according to ρ'. This is a contradiction. □

Lemma 3.3. *If a game G is strongly connected and belongs to $\Sigma_{n+1} \cap \Pi_{n+1}$ then either it belongs to Σ_n, or it belongs to Π_n.*

Proof. If G belongs to $\Sigma_{n+1} \cap \Pi_{n+1}$ then there exist two mappings ρ and ρ', equivalent to ρ^G w.r.t. G, whose images are respectively included in $\{m-n, \ldots, m\}$ and $\{m'-n, \ldots, m'\}$ where m is odd and m' is even.

Assume that there are two positions $p, p' \in Pos^G_{E,A}$ such that $\rho(p) = m$ and $\rho'(p') = m'$. Since G is strongly connected, there exists a non empty path γ from p to p' and a non empty path γ' from p' to p. The maximal value of ρ (resp. ρ') which occurs infinitely often in the path $(\gamma\gamma')^\omega$ is m (resp. m'). Therefore this infinite path is a win for Adam according to ρ and a win for Eva according to ρ', a contradiction as ρ and ρ' are equivalent.

It follows that either ρ never takes the value m on $Pos^G_{E,A}$ or ρ' never takes the value m' on $Pos^G_{E,A}$. In the first case $\rho(Pos^G_{E,A}) \subseteq \{m-n, \ldots, m-1\}$ and $G \in \Pi_n$. In the second case $\rho'(Pos^G_{E,A}) \subseteq \{m'-n, \ldots, m'-1\}$ and $G \in \Sigma_n$. □

Corollary 3.4. *If a game G belongs to Σ_{n+1} and to Π_{n+1}, then it belongs to $Comp(\Sigma_n, \Pi_n)$.*

Proof. If $n = 0$ then this is lemma 3.2. Otherwise we can construct G from its maximal strongly connected components G_i by means of a sequences of substitutions. According to lemma 3.3, each of the G_i is either in Σ_n or in Π_n. Therefore $G \in Comp(\Sigma_n, \Pi_n)$. □

Thus we have argued that the equality (1) holds at the syntactic level. In the introduction we have stressed that the relevant question is whether such equality holds with respect to the given interpretation of all complete lattices. By the characterization in [20], this is the same as asking whether such equation holds up to the equivalence relation $\sim$ induced by the preorder $\leq$.

Definition 3.5. Let $G \in \mathcal{G}$ and say that $G \in \mathcal{S}_n$ if there exists a $G' \in \Sigma_n$ such that $G \sim G'$. Similarly, say that $G \in \mathcal{P}_n$ if there exists a $G' \in \Pi_n$ such that $G \sim G'$, and that $G \in \mathcal{C}_n$ if there exists a $G' \in Comp(\Sigma_n, \Pi_n)$ such that $G \sim G'$.

The *ambiguous class* $\mathcal{D}_n$ is simply the intersection of $\mathcal{P}_n$ and $\mathcal{S}_n$. The main result of this paper is the following:

Theorem 3.6. *The ambiguous class $\mathcal{D}_{n+1} = \mathcal{P}_{n+1} \cap \mathcal{S}_{n+1}$ and the class $\mathcal{C}_n$ are equal, for every $n \geq 0$.*

The relation $\mathcal{C}_n \subseteq \mathcal{P}_{n+1} \cap \mathcal{S}_{n+1}$ immediately follows from the definition of the classes $\mathcal{C}_n, \mathcal{S}_{n+1}, \mathcal{P}_{n+1}$ and by the relation (1). For the converse it is enough to prove the following Proposition.

Proposition 3.7. *Let G and H be games in Π_{n+1} and Σ_{n+1}, respectively, and suppose that $G \leq H$. Then there exists a $K \in Comp(\Sigma_n, \Pi_n)$ such that $G \leq K$ and $K \leq H$.*

Indeed, if $G' \in \mathcal{S}_{n+1} \cap \mathcal{P}_{n+1}$, then let $G \in \Pi_{n+1}$ and $H \in \Sigma_{n+1}$, such that $G' \sim G \sim H$. If K is as in the statement of Proposition 3.7, then the relations

$$G' \leq G \leq K \leq H \leq G'$$

exhibit G' as a member of $\mathcal{C}_n$.

Proof (of Proposition 3.7). Let us fix $G \in \Pi_{n+1}$ and $H \in \Sigma_{n+1}$, thus we shall assume that $\rho^G(Pos^G_{E,A}) \subseteq \{m-n, \ldots, m\}$ where m is even and that $\rho^H(Pos^H_{E,A}) \subseteq \{m'-n, \ldots, m'\}$ with m' odd. We also assume that $G \leq H$ and fix a winning strategy for Mediator in the game $\langle\!\langle G, H \rangle\!\rangle$ from position $(p^G_\star, O, p^H_\star)$. This game is almost[6] a game whose set of infinite winning plays is described by

[6] The winning condition can be described using Rabin pairs on the edges.

a Rabin acceptance condition. Thus, if Mediator has a winning strategy in this game, then he has a deterministic bounded memory winning strategy as well. Therefore we shall assume that the fixed winning strategy is deterministic and has a bounded memory. We shall represent it as the tuple $\langle S, U, s_\star, \psi\rangle$, where $\langle S, U, s_\star\rangle$ is a finite pointed graph, with set of memory states S, set of update transitions U, and an initial state $s_\star$; ψ is an homomorphism of graphs from $\langle S, U, s_\star\rangle$ to the graph of $\langle\!\langle G, H\rangle\!\rangle$ (mapping every memory state to a position and an update transition to a move) with the following properties:

- $\psi(s_\star) = (p^G_\star, O, p^H_\star)$,
- if $s \in S$ and $\psi(s) = (g, O, h)$ is an Opponent's position, then for every move $(g, O, h) \to (g', M, h')$ there exists a unique s' such that $s \to s'$ and $\psi(s') = (g', M, h')$,
- if $s \in S$ and $\psi(s) = (g, M, h)$ is a Mediator's position, then there exists a unique transition $s \to s'$,
- if $s_0 \to s_1 \to \dots$ is an infinite path in the graph $\langle S, U\rangle$, then the infinite play $\psi(s_0) \to \psi(s_1) \to \dots$ is a win for Mediator.

Recall from 1.1 the definition of the predecessor game G^-. In particular, recall that $Top^G = \{\, g \in Pos^G_{E,A} \,|\, \rho^G(g) = Max^G \,\}$ and that $Top^H = \{\, h \in Pos^H_{E,A} \,|\, \rho^H(h) = Max^H \,\}$. Observe that, for the games G and H under consideration, G^- belongs to Σ_n and H^- belongs to Π_n. Intuitively, our next goal is to show that we can completely decompose the given winning strategy into a collection of local strategies that Mediator can play either in $\langle\!\langle G, H^-\rangle\!\rangle$, or in $\langle\!\langle G, H'\rangle\!\rangle$ for some game H' of the form $\bigwedge_I$, or in $\langle\!\langle G^-, H\rangle\!\rangle$ or in some $\langle\!\langle G', H\rangle\!\rangle$ for some game G' of the form $\bigvee_I$.

We shall denote by $[\,s\,]$ the maximal strongly connected component of the graph $\langle S, U\rangle$ to which s belongs. We observe that the following exhaustive and exclusive cases arise:

(Ac) The component $[\,s\,]$ is reduced to the singleton $\{s\}$. Observe that we cannot have a transition $s \to s$ as the graph of $\langle\!\langle G, H\rangle\!\rangle$ is bipartite. Therefore the component $[\,s\,]$ is acyclic.

(Cy) The component $[\,s\,]$ is cyclic (and contains at least two elements). We have the following subcases:

(CyA) The projection of $[\,s\,]$ onto H is stuck and belongs to Adam: let $s_1, s_2 \in [\,s\,]$ be such that $s_1 \to s_2$ and let $\psi(s_i) = (g_i, P_i, h_i)$ for $i = 1, 2$; then $h_1 = h_2 \in Pos^H_A$ and the move $(g_1, P_1, h_1) \to (g_2, P_2, h_1)$ is either a left move (i.e. $(g_1, g_2) \in M^G$) or it is a silent move.

(CyE) The projection of $[\,s\,]$ onto G is stuck and belongs to Eva: the formal definition is obtained by exchanging H with G and Adam with Eva in the definition of (CyA).

The previous conditions do not hold and:

(CyG) The projection of $[\,s\,]$ onto G contains a visit to Top^G: there exists an $s' \in [\,s\,]$ such that $\psi(s') = (g', P', h')$ and $\rho^G(g') = Max^G$.

(CyH) The projection of $[\,s\,]$ onto H contains a visit to Top^H: there exists an $s' \in [\,s\,]$ such that $\psi(s') = (g', P', h')$ and $\rho^H(h') = Max^H$.

(CyN) None of the previous conditions hold. In particular, for all $s' \in [\,s\,]$, if $\psi(s') = (g', P', h')$, $g' \in Pos^G_{E,A}$ implies $\rho^G(g') < Max^G$ and $h' \in Pos^H_{E,A}$ implies $\rho^G(h') < Max^H$.

The reader should verify that the above cases are indeed disjoint. To see that (CyA) and (CyE) are disjoint, observe that a proper cycle in the graph of $\langle\!\langle G, H\rangle\!\rangle$ cannot be stuck both on G and on H. To see that (CyG) and (CyH) are disjoint consider a maximal strongly connected component $[\,s\,]$ that visits both Top^G and Top^H, and a path γ that visits all the states in $[\,s\,]$. The unique way the path $\psi(\gamma^\omega)$ can be a win in the game $\langle\!\langle G, H\rangle\!\rangle$ for Mediator is that the play $\psi(\gamma)$ is stuck on H on an Adam's position, in which case $[\,s\,]$ satisfies (CyA), or that this play is stuck on G on an Eva's position, in which case $[\,s\,]$ satisfies (CyE).

Definition 3.8. We say that $s \mapsto s'$ if and only if there exists a path $s = s_0 \to s_1 \to \ldots \to s_n = s'$, but s' does not belong to the strongly connected component of s.

Clearly, the relation $s \mapsto s'$ is irreflexive and acyclic, and therefore well founded.

Lemma 3.9. *Let $s \in S$ and $\psi(s) = (g, P, h)$, where $P \in \{O, M\}$. Suppose that the strongly connected component $[\,s\,]$ is of type (CyG) or of type (CyN). If for each $h' \in Top^H$ there exists $\kappa(h')$ such that $G_{g'} \leq \kappa(h')$ whenever $s \mapsto s'$ and $\psi(s') = (g', P', h')$, then*

$$G_g \leq H^-_h[\kappa(h')/y_{h'}]_{h' \in Top^H}\,.$$

Of course there is a dual lemma if the stronlgy connected component $[\,s\,]$ is of type (CyH); we leave the reader to formulate it. Observe that in order to form a collection $\{\kappa(h')\}$ satisfying the hypothesis of the lemma, it is enough to let $\kappa(h') = \bigvee_\emptyset$ if there is no s' such that $s \mapsto s'$ and $\psi(s') = (g', P', h')$.

Proof (of lemma 3.9). The positions of the game $\langle\!\langle G, H^-_h[\kappa(h')/y_{h'}]_{h' \in Top^H}\rangle\!\rangle$ form a set which is the disjoint union of a component $Pos^G_{E,A,D} \times \{M, O\} \times (Pos^H_{E,A,D} - Top^H)$ and of components $Pos^G_{E,A,D} \times \{M, O\} \times Pos^{\kappa(h')}_{E,A,D}$ for $h' \in Top^H$. Moreover, in the latter components, the game is exactly as in $\langle\!\langle G, \kappa(h')\rangle\!\rangle$.

Mediator can use the strategy S from position $\psi(s)$ on the first component $Pos^G_{E,A,D} \times \{M, O\} \times (Pos^H_{E,A,D} - Top^H)$, as long as the strategy does not suggest a move $(g', P, h) \to (g', P', h')$ for some $h' \in Top^H$. If this is the case and if s' is the state of the strategy that lifts (g', P', h'), then $s \mapsto s'$, because $[\,s\,]$ cannot contain a visit to Top^H. Hence, by assumption, there is a winning strategy in the game $\langle\!\langle G_{g'}, \kappa(h')\rangle\!\rangle$ from both positions $(g', O, p^{\kappa(h')}_\star)$ and $(g', M, p^{\kappa(h')}_\star)$, by lemma 2.3. The move $(g', P, h) \to (g', P', h')$ becomes a move to $(g', P', p^{\kappa(h')}_\star)$ in $\langle\!\langle G, \kappa(h')\rangle\!\rangle$ and Mediator can continue with a winning strategy from the latter position. □

We complete now the proof of Proposition 3.7 by proving the following stronger claim:

Claim. For each $s \in S$ such that $\psi(s) = (g, P, h)$ there is a game K_s in the class $Comp(\Sigma_n, \Pi_n)$ such that $G_g \leq K_s \leq H_h$.

The proof is by induction on the well founded relation $\mapsto$ and it is subdivided into cases, according to the type of the strongly connected component $[\, s \,]$.

We suppose first that the type of $[\, s \,]$ is **(Ac)**, so that if $s \to s'$ then $s \mapsto s'$. Observe that if $s \to s'$ is a transition lifting a silent move of the form

$$(g, O, h) \to (g, M, h) \quad (g, M, h) \to (g, O, h)$$

then there is essentially nothing to prove: we can let $K_s = K_{s'}$ since by the induction hypothesis $G_g \leq K_{s'} \leq H_h$.

If $g \in Pos_E^G$ and $P = O$, then for each move $(g, g') \in M^G$ there is a move $(g, O, h) \to (g', M, h)$ and a lifting $s \to s(g')$ of this move. By the induction hypothesis there are $K_{s(g')} \in Comp(\Sigma_n, \Pi_n)$ such that $G_{g'} \leq K_{s(g')} \leq H_h$. We can let $K_s = \bigvee_{(g,g') \in M^G} K_{s(g')} \in Comp(\Sigma_n, \Pi_n)$, it follows that

$$G_g \sim \bigvee_{(g,g') \in M^G} G_{g'} \leq \bigvee_{(g,g') \in M^G} K_{s(g')} \leq H_h \,.$$

Assume now that $g \in Pos_A^G$, $P = M$, and that the unique transition $s \to s'$ of the strategy is suggesting a move of the form $(g, M, h) \to (g', M, h)$ for some $(g, g') \in M^G$. We let $K_s = K_{s'} \in Comp(\Sigma_n, \Pi_n)$, and knowing that $G_{g'} \leq K_{s'} \leq H_h$ we derive

$$G_g \sim \bigwedge_{(g,g') \in M^G} G_{g'} \leq G_{g'} \leq K_s \leq H_h \,.$$

We can use a dual argument if $h \in Pos_A^H$ and $P = O$ or if $h \in Pos_E^H$ and $P = M$. If $g \in Pos_D^G$ and $h \in Pos_D^H$, then we let K_s be the game with only one position labeled by $\lambda^G(g)$.

We suppose now that the type of $[\, s \,]$ is **(CyA)**. Observe that if $s' \in [\, s \,]$ and $\psi(s') = (g', O, h)$ is an Opponent position, then for each move $(h, h') \in M^H$ there is a move $(g', O, h) \to (g', M, h')$ in $\langle\!\langle G, H \rangle\!\rangle$ and a lifting of this move $s' \to s'(h')$ in $\langle S, U \rangle$. By definition of the type (CyA), $s'(h') \notin [\, s \,]$, hence there exists a $K_{s'(h')}$ such that $G_{g'} \leq K_{s'(h')} \leq H_{h'}$. We can let $K_{s'} = \bigwedge_{(h,h') \in M^H} K_{s'(h')}$ since this game belongs to $Comp(\Sigma_n, \Pi_n)$ and

$$G_{g'} \leq \bigwedge_{(h,h') \in M^H} K_{s'(h')} \leq \bigwedge_{(h,h') \in M^H} H_{h'} \sim H_h \,.$$

If $s' \in [\, s \,]$ and $\psi(s') = (g', M, h)$, then there is a unique transition $s' \to s''$. If $s \mapsto s'$ then we can use the inductive hypothesis; otherwise, if $s'' \in [\, s \,]$, we observe that $\psi(s'') = (g'', O, h)$ is an Opponent position and that we have described how to construct $K_{s''}$ satisfying the claim in the previous parargraph. As the relation $G_{g'} \leq G_{g''}$ holds, we can let $K_{s'} = K_{s''}$, since

$$G_{g'} \leq G_{g''} \leq K_{s''} \leq H_h \,.$$

We suppose now that the type of $[\,s\,]$ is either **(CyG)** or **(CyH)**. For each $h' \in Top^H$ let

$$\kappa(h') = \bigvee_{\substack{s \,\mapsto\, s' \\ \psi(s') \,=\, (g', P', h')}} K_{s'}$$

where the $K_{s'} \in Comp(\Sigma_n, \Pi_n)$ have been previously constructed and satisfy the relation $G_{g'} \leq K_{s'} \leq H_{h'}$. Observe that $G_{g'} \leq \kappa(h')$ whenever $s \mapsto s'$ and $\psi(s') = (g', P', h')$, therefore by lemma 3.9 the relation

$$G_g \leq H_h^-[\kappa(h')/y_{h'}]_{h' \in Top^H}$$

holds. Also, we have $\kappa(h') \leq H_{h'}$ for all $h' \in Top^H$ and therefore

$$H_h^-[\kappa(h')/y_{h'}]_{h' \in Top^H} \leq H_h^-[H_{h'}/y_{h'}]_{h' \in Top^H}$$

where the last game is clearly equivalent to H_h. If we let K_s be the game $H_h^-[\kappa(h')/y_{h'}]_{h' \in Top^H}$, then K_s belongs to $Comp(\Sigma_n, \Pi_n)$, since $H_h^- \in \Pi_n$, $\Pi_n \subseteq Comp(\Sigma_n, \Pi_n)$, and for all $h' \in Top^H$ $\kappa(h') \in Comp(\Sigma_n, \Pi_n)$. Moreover we have shown that $G_g \leq K_s \leq H_h$.

We can use dual arguments if the strongly connected component is of type **(CyE)** or **(CyH)**; therefore the claim holds for every $s \in S$ and for $s_\star \in S$ in particular. As we have $\psi(s_\star) = (p_\star^G, O, p_\star^H)$, the relations

$$G = G_{p_\star^G} \leq K_{s_\star} \leq H_{p_\star^H} = H\,,$$

prove Proposition 3.7. □

Finally we remark that if there exists a bounded memory winning strategy in the game $\langle\!\langle G, H \rangle\!\rangle$ for Mediator, then there exists a winning strategy for Mediator of size $|G| \times |H|$, where $|G| = \operatorname{card} Pos_{E,A,D}^G + \operatorname{card} M^G$ is the size of a game G. This follows from considerations developed in [8]. Thus effective bounds to construct K such that $G \leq K \leq H$ can be extracted out of this information.

References

[1] A. Arnold. The μ-calculus alternation-depth hierarchy is strict on binary trees. *Theor. Inform. Appl.*, 33(4-5):329–339, 1999.

[2] A. Arnold and D. Niwiński. Fixed point characterization of weak monadic logic definable sets of trees. In *Tree automata and languages (Le Touquet, 1990)*, volume 10 of *Stud. Comput. Sci. Artificial Intelligence*, pages 159–188. North-Holland, Amsterdam, 1992.

[3] A. Arnold and D. Niwiński. *Rudiments of μ-calculus*, volume 146 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, 2001.

[4] A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial information. To appear in Theoret. Comput. Sci., 2002.

[5] A. Blass. A game semantics for linear logic. *Ann. Pure Appl. Logic*, 56(1-3):183–220, 1992.
[6] J. C. Bradfield. The modal mu-calculus alternation hierarchy is strict. *Theoret. Comput. Sci.*, 195(2):133–153, 1998.
[7] G. D'Agostino and M. Hollenberg. Logical questions concerning the μ-calculus: interpolation, Lyndon and łoś-Tarski. *J. Symbolic Logic*, 65(1):310–332, 2000.
[8] M. Dziembowski, S. Jurdziński and I. Walukiewicz. How much memory is needed to win infinite games? In *Twelth Annual IEEE Symposium on Logic in Computer Science*, pages 99–110. IEEE Computer Society Press, 1997.
[9] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model checking for the μ-calculus and its fragments. *Theoret. Comput. Sci.*, 258(1-2):491–522, 2001.
[10] R. Freese, J. Ježek, and J. B. Nation. *Free lattices.* American Mathematical Society, Providence, RI, 1995.
[11] A. Joyal. Free lattices, communication and money games. In *Logic and scientific methods (Florence, 1995)*, pages 29–68. Kluwer Acad. Publ., Dordrecht, 1997.
[12] A. S. Kechris. *Classical descriptive set theory*, volume 156 of *Graduate Texts in Mathematics.* Springer-Verlag, New York, 1995.
[13] G. Lenzi. A hierarchy theorem for the μ-calculus. In *Automata, languages and programming (Paderborn, 1996)*, volume 1099 of *Lecture Notes in Comput. Sci.*, pages 87–97. Springer, Berlin, 1996.
[14] A. W. Mostowski. Regular expressions for infinite trees and a standard form of automata. In *Computation theory (Zaborów, 1984)*, volume 208 of *Lect. Notes in Comput. Sci.*, pages 157–168. Springer, Berlin, 1985.
[15] D. Niwinski. On fixed-point clones. In *Automata, languages and programming*, volume 226 of *Lecture Notes in Comput. Sci.*, pages 464–473, 1986.
[16] D. Niwiński. Fixed point characterization of infinite behavior of finite-state systems. *Theoret. Comput. Sci.*, 189(1-2):1–69, 1997.
[17] D. Niwiński and I. Walukiewicz. Relating hierarchies of word and tree automata. In *STACS 98 (Paris, 1998)*, volume 1373 of *Lecture Notes in Comput. Sci.*, pages 320–331. Springer, Berlin, 1998.
[18] L. Santocanale. The alternation hierarchy for the theory of μ-lattices. *Theory and Applications of Categories*, 9:166–197, January 2002.
[19] L. Santocanale. A calculus of circular proofs and its categorical semantics. In *FOSSACS02*, volume 2303 of *Lecture Notes in Comput. Sci.*, pages 357–371, 2002.
[20] L. Santocanale. Free μ-lattices. *J. Pure Appl. Algebra*, 168(2-3):227–264, 2002.
[21] L. Santocanale. Parity games and μ-bicomplete categories. *Theor. Inform. Appl.*, (36):195–227, 2002.
[22] G. Takeuti. *Proof theory.* North-Holland Publishing Co., Amsterdam, 1975. Studies in Logic and the Foundations of Mathematics, Vol. 81.
[23] W. Thomas. Languages, automata, and logic. In Rozenberg G. and Salomaa A., editors, *Handbook of Formal Language Theory*, volume III, pages 389–455. Springer-Verlag, New York, 1996.

Parameterized Verification by Probabilistic Abstraction*

Tamarah Arons[1], Amir Pnueli[1], and Lenore Zuck[2]

[1] Weizmann Institute of Science, Rehovot, Israel,
{amir,tamarah}@wisdom.weizmann.ac.il
[2] New York University, New York,
zuck@cs.nyu.edu

Abstract. The paper studies automatic verification of liveness properties with probability 1 over parameterized programs that include probabilistic transitions, and proposes two novel approaches to the problem. The first approach is based on a *Planner* that occasionally determines the outcome of a finite sequence of "random" choices, while the other random choices are performed non-deterministically. Using a Planner, a probabilistic protocol can be treated just like a non-probabilistic one and verified as such. The second approach is based on γ-*fairness*, a notion of fairness that is sound and complete for verifying simple temporal properties (whose only temporal operators are $\diamondsuit$ and $\square$) over finite-state systems. The paper presents a symbolic model checker based on γ-fairness. We then show how the network invariant approach can be adapted to accommodate probabilistic protocols. The utility of the Planner approach is demonstrated on a probabilistic mutual exclusion protocol. The utility of the approach of γ-fairness with network invariants is demonstrated on Lehman and Rabin's Courteous Philosophers algorithm.

1 Introduction

Probabilistic elements have been introduced into concurrent systems in the early 1980s to provide solutions (with high probability) to problems that do not have deterministic solutions. Among the pioneers of probabilistic protocols were ([LR81, Rab82]). One of the most challenging problems in the study of probabilistic protocols has been their formal verification. While methodologies for proving safety (invariance) properties still hold for probabilistic protocols, formal verification of their liveness properties has been, and still is, a challenge. The main difficulty stems from the two types of nondeterminism that occur in such programs: Their asynchronous execution, that assumes a potentially adversarial (though somewhat fair) scheduler, and the nondeterminism associated with the probabilistic actions, that assumes an even-handed coin-tosser.

It had been realized that if one only wants to prove that a certain property is *P-valid*, i.e., holds with probability 1 over all executions of a system, this can be accomplished,

* This research was supported in part by ONR grant N00014-99-1-0131, and the John von Neumann Minerva Center for Verification of Reactive Systems.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 87–102, 2003.

for finite-state systems, in a manner that is completely independent of the precise probabilities. Decidability of P-validity had been first established in [HSP82] for termination properties over finite-state systems, using a methodology that is graph-theoretic in nature. The work in [PZ86b] extends the [HSP82] method and presents deductive proof rules for proving P-validity for termination properties of finite-state program. The work in [PZ93] presents sound and complete methodology for establishing P-validity of general temporal properties over probabilistic systems, and [VW86, PZ93] describe explicit-state model checking procedures for the finite-state case.

The emerging interest in embedded systems brought forth a surge of research in automatic verification of parameterized systems, that, having unbounded number of states, are not easily amenable to model checking techniques. In fact, verification of such systems is known to be undecidable [AK86]. Much of the recent research has been devoted to identifying conditions that enable automatic verification of such systems, and abstraction tools to facilitate the task (e.g., [KP00, APR$^+$01, EN95, EN96, EK00, KPSZ02].)

One of the promising approaches to the uniform verification of parameterized systems is the method of *network invariants*, first mentioned in [BCG86, SG89], further developed in [WL89] (who also coined the name "network invariant"), and elaborated in [KM95] into a working method. In [KP00, KPSZ02] we extended the approach by using a notion of abstraction that takes into account fairness properties. The approach was developed into a working method and implemented on the Weizmann Institute Temporal Logic Verifier TLV [PS96].

Another promising approach to the uniform verification of parameterized systems is the method of *counter abstraction*: Given a parameterized system with finitely many local states, a concrete state is abstracted by counting, for each possible local state, the minimum between 2 and the number of processes with that local state. Traditionally, counter abstraction is used for proving safety properties of parameterized systems (e.g., [Lub84].) More recently ([BLS01, PXZ02]) it was applied also to the verification of liveness properties; [PXZ02] employs explicit abstraction of fairness requirements.

Since many of the probabilistic protocols that have been proposed and studied (e.g., [LR81, Rab82, CLP84]) are parameterized, a naturally arising question is whether we can combine automatic verification tools of parameterized systems with those of probabilistic ones.

In this paper we propose two novel approaches to the problem. The first is based on *Planners* and the second on the notion of γ*-fairness* introduce in [ZPK02]. When activated, a planner pre-determines the results of a the next k consecutive "random" choices, allowing these next choices to be performed in a completely non-deterministic manner. The approach is sound for finite-state systems: if there is a planner such that a temporal property holds over all computations of the (non-probabilistic) program that activates the planner infinitely often, then the property is P-valid. To deal with parameterized systems, we abstract a version of the system which activates the planner infinitely many times.

The notion of γ*-fairness* is a notion of fairness that is sound and complete for verifying simple temporal properties (whose only temporal operators are $\diamondsuit$ and $\square$) over finite-state systems. We devised a *symbolic* model checking algorithm based on

γ-fairness that automatically verifies simple temporal properties of finite-state systems. The algorithm was implemented using TLV. We also extended the network invariant method to apply to γ-fairness, obtaining a method for verifying P-validity over parameterized systems.

Whereas we consider only P-validity, the PRISM probabilistic model checker [KNP02], based on Markov chains and processes, allows the user to verify that a property holds with arbitrary probability, and not just probability 1. However, PRISM does not support the uniform verification of parameterized systems, but rather the verification of individual system configurations. Thus, while PRISM was used to automatically verify the [PZ86b] mutual exclusion protocol for $N = 10$, we (in Section 5) automatically verify it for *every* $N > 1$.

The paper is organized as follows: In Section 2 we describe our formal model, *probabilistic discrete systems* (PDS), which is a fair discrete system augmented with probabilistic requirements. We then define the notion of P-validity over PDSs. We also briefly describe the programming language (SPL augmented with probabilistic goto statements) that we use in our examples and its relation to PDS. In Section 3 we introduce our two new methods for proving P-validity over finite-state systems: The Planner approach, and γ-fairness. We also introduce SYMPMC, the symbolic model checker based on γ-fairness. In Section 4 we describe our model for (fully symmetric) parameterized systems, the method of counter abstraction, and the method of network invariants extended to deal with γ-fairness. Section 5 contains two examples: An automatic P-validity proof of the liveness property of parameterized probabilistic mutual exclusion algorithm [PZ86b] that uses a Planner combined with counter-abstraction, and an automatic proof of P-validity of the individual liveness of the Courteous Philosophers algorithm [LR81] using SYMPMC and Network Invariants. The mutual exclusion example is, to our knowledge, the first formal and automatic verification of this protocol (and protocols similar to it.) The Courteous Philosopher example was proven before in [KPSZ02]. However, there we converted the protocol to a non-probabilistic one with compassion requirements, that had to be devised manually and checked separately, and only then applied the network invariant abstraction. Here the abstraction is from a probabilistic, into a probabilistic, protocol. The advantage of this method is that it does not require the user to compile the list of compassion requirements, nor for the compassion requirements to be checked. Since checking the probabilistic requirements directly is more efficient than checking multiple compassion requirements, the run times are significantly faster (speedups of 50 to 90 percent.) We conclude in Section 6.

2 The Framework

As a computational model for reactive systems we take the model of *fair discrete system* (FDS) [KP00], which is a slight variation on the model of *fair transition system* [MP95], and add *probabilistic requirements* that describe the outcomes of probabilistic selections. We first describe the formal model and the notion of *P-validity*—validity with probability 1. We then briefly describe an a simple programming language that allows for probabilistic selections.

2.1 Probabilistic Discrete Systems

In the systems we are dealing with, all probabilities are bounded from below. In addition, the only properties we are concerned with are temporal formulae which hold with probability 1. For simplicity of presentation, we assume that all probabilistic choices are *binary with equal probabilities*. These restrictions can be easily removed without impairing the results or methods presented in this paper.

A *probabilistic discrete system (*PDS*)* $S : \langle V, \Theta, \rho, \mathcal{P}, \mathcal{J}, \mathcal{C} \rangle$ consists of the following components:

- V: A finite set of typed *system variables*, containing data and control variables. A state s is an assignment of type-compatible values to the system variables V. For a set of variables $U \subseteq V$, we denote by $s[U]$ the set of values assigned by state s to the variables U. The set of states over V is denoted by Σ. We assume that Σ is finite.
- Θ: An *initial condition* – an *assertion* (first-order state formula) characterizing the initial states.
- ρ: A *transition relation* – an assertion $\rho(V, V')$, relating the values V of the variables in state $s \in \Sigma$ to the values V' in a ρ-successor state $s' \in \Sigma$.
- $\mathcal{P}$: A finite set of *probabilistic selections*, each is a triplet (r, t_1, t_2) where r, t_1 and t_2 are assertions. Each such triplet denotes that t_1- and t_2-states are the possible outcomes of a probabilistic transition originating at r-states. We require that for every s and s' such that s' is a ρ-successor of s, there is at most one probabilistic selection $(r, t_1, t_2) \in \mathcal{P}$ and one $i \in \{1, 2\}$ such that s is an r-state and s' is a t_i-state. Thus, given two states, there is at most a single choice out of a single probabilistic selection that can lead from one to the other.
- $\mathcal{J}$: A set of *justice* (*weak fairness*) *requirements*, each given as an assertion. They guarantee that every computation has infinitely many J-states, for every $J \in \mathcal{J}$.
- $\mathcal{C}$: A set of *compassion* (*strong fairness*) *requirements*, each is a pair of assertions. They guarantee that every computation has either finitely many p-states or infinitely many q-states, for every $(p, q) \in \mathcal{C}$.

We require that every state $s \in \Sigma$ has *some* transition enabled on it. This is often ensured by requiring ρ to include the disjunct $V = V'$ which represents the *idle* transition.

Let S be an PDS for which the above components have been identified. We define a *computation tree* of S to be an infinite tree whose nodes are labeled by Σ defined as follows:

- *Initiality:* The root of the tree is labeled by an initial state, i.e., by a Θ-state.
- *Consecution:* Consider a node n labeled by a state s. Then one of the following holds:
  1. n has two children, n_1 and n_2, labeled by s_1 and s_2 respectively, such that for some $(r, t_1, t_2) \in \mathcal{P}$, s is an r-state, and s_1 and s_2 are t_1- and t_2-states respectively.
  2. n has a single child n' labeled by s', such that s' is a ρ-successor of s and for no $(r, t_1, t_2) \in \mathcal{P}$ is it the case that s is an r-state and s' is a t_i-state for some $i \in \{1, 2\}$.

Consider an infinite path $\pi : s_0, s_1, \ldots$ in a computation tree. The path π is called *just* if it contains infinitely many occurrences of J-states for each $J \in \mathcal{J}$. Path π is called *compassionate* if, for each $(p, q) \in \mathcal{C}$, π contains only finitely many occurrences of p-states or π contains infinitely many occurrences of q-states. The path π is called *fair* if it is both just and compassionate.

A computation tree induces a probability measure over all the infinite paths that can be traced in the tree, the edges leading from a node with two children have each probability 0.5, and the others have probability 1. We say that a computation tree is *admissible* if the measure of fair paths in it is 1. Following [PZ93], we say that a temporal property φ is *P-valid* over a computation tree if the measure of paths in the tree that satisfy φ is 1. (See [PZ93] for a detailed description and definition of the measure space.) Similarly, φ is *P-valid over the* PDS S if it is P-valid over every admissible computation tree of S.

Note that when S is non-probabilistic, that is, when $\mathcal{P}$ is empty, then the notion of P-validity over S coincides with the usual notion of validity over S.

2.2 Probabilistic SPL

All our concrete examples are given in SPL (Simple Programming Language), which is used to represent concurrent programs (e.g., [MP95, MAB+94]). Every SPL program can be compiled into a PDS in a straightforward manner. In particular, every statement in an SPL program contributes a disjunct to the transition relation. For example, the assignment statement "$\ell_0 : \mathtt{x} := \mathtt{y} + 1; \ell_1 :$ " can be executed when control is at location ℓ_0. When executed, it assigns the value of $y + 1$ to the variable x while control moves from ℓ_0 to ℓ_1. This statement contributes to the transition relation, in the PDS that describes the program, the disjunct $at\_\ell_0 \ \wedge \ at'\_\ell_1 \ \wedge \ x' = y + 1 \ \wedge \ pres(V - \{x, \pi\})$ (where for a set $U \subseteq V$, $pres(U) = \bigwedge_{u \in U} u' = u$) and nothing to the probabilistic requirements. The predicates at_ℓ_0 and at'_ℓ_1 stand, respectively, for the assertions $\pi = 0$ and $\pi' = 1$, where π is the control variable denoting the current location within the process to which the statement belongs (program counter).

In order to represent probabilistic selections, we augment SPL by probabilistic goto statements of the form

$$\ell_0 : \ \textbf{pr\_goto} \ \{\ell_1, \ell_2\}$$

which adds the disjunct $at\_\ell_0 \ \wedge \ (at'\_\ell_1 \vee at'\_\ell_2) \ \wedge \ pres(V - \{\pi\})$ to the transition relation and the triplet $(at\_\ell_0, at\_\ell_1, at\_\ell_2)$ to the set of probabilistic requirements. Note that, since we allow stuttering (idling), we lose no generality by allowing only probabilistic goto's, as opposed to more general probabilistic assignments.

3 Automatic Verification of Finite-State PDSs

Automatic verification of finite-state PDSs has been studied in numerous works e.g., [VW86, PZ86a, KNP02]. Here we propose two new approaches to automatic verification of P-validity of finite-state PDSs, both amenable to dealing with parameterized (infinite-state) systems.

3.1 Automatic Verification Using Planners

A *planner* transforms a probabilistic program Q into a non-probabilistic program $Q_{_T}$ by pre-determining the results of a the next k consecutive "random" choices, allowing these next choices to be performed in a completely non-deterministic manner. The transformation is such that, for every temporal formula φ over (the variables of) Q, if φ is valid over $Q_{_T}$, then it is P-valid over Q. Thus, the planner transformation converts a PDS into an FDS , and reduces P-validity into validity. The transformation is based on the following consequence of the Borel-Cantelli lemma [Fel68]:

> Let $b_1, \ldots, b_k$ be a sequence of values, $b_i \in \{1, 2\}$, for some fixed k. Let σ be a computation of the program Q which makes infinitely many probabilistic selections. Then, with probability 1, σ contains infinitely many segments containing precisely k probabilistic choices in which these choices follow the pattern $b_1, \ldots, b_k$.

The transformation from Q to $Q_{_T}$ can be described as follows: Each probabilistic statement "ℓ: **pr_goto** $\{\ell_1, \ell_2\}$" in Q is transformed into:

if $\mathit{consult}_\ell > 0$
then $\left[\begin{array}{l} \mathit{consult}_\ell := \mathit{consult}_\ell - 1 \\ \textbf{if } \mathit{planner}_\ell \textbf{ then goto } \ell_1 \textbf{ else goto } \ell_2 \end{array}\right]$
else goto $\{\ell_1, \ell_2\}$

Thus, whenever counter $\mathit{consult}_\ell$ is positive, the program invokes the boolean-valued function $\mathit{planner}_\ell$ which determined whether the program should branch to ℓ_1 or to ℓ_2. Each such "counselled" branch decrements the counter $\mathit{consult}_\ell$ by 1. When the counter is 0, the branching is purely non-deterministic. The function $\mathit{planner}_\ell$ can refer to all available variables. Its particular form depends on the property φ, and it is up to the user of this methodology to design a planner appropriate for the property at hand. This may require some ingenuity and a thorough understanding of the analyzed program.

Finally, we augment the system with a parallel process, the *activator*, that non-deterministically sets all $\mathit{consult}_\ell$ variables to a constant value k. We equip this process with a justice requirement that guarantees that it replenishes the counters (activates the planners) infinitely many times. A proof for the soundness of the method is in Appendix A.

Example 1. Consider Program *up-down* in Fig. 1 in which two processes increment and decrement $y \in [0..4]$.

To establish the P-validity of $\varphi : \Box\Diamond(y = 0)$, we can take $k = 4$ and define both $\mathit{planner}_{\ell_0}$ and $\mathit{planner}_{m_0}$ to always yield 0, thus consistently choosing the second (decrementing) mode.

3.2 SYMPMC: A Symbolic Probabilistic Model Checker

While the planner strategy is applicable for many systems, there are cases of parameterized systems (defined in Section 4) for which it cannot be applied. (See Section 5.2

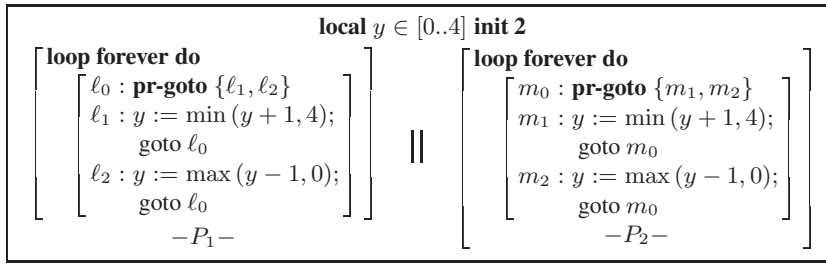


Fig. 1. Program *up-down*

for an example.) As an alternative method we describe SYMPMC, a symbolic model checker that verifies P-validity of *simple* temporal properties—properties whose only temporal operators are $\diamondsuit$ and $\square$—over finite state PDSs.

The motivation leading to SYMPMC has been the idea that, since all the probabilities of a finite PDS are bounded from below, the definition of P-validity that directly deals with measure spaces can be replaced with some simpler notions of fairness. This was first done in [Pnu83], where *extreme-fairness* was introduced, a notion of fairness that is sound and incomplete for proving P-validity. The work in [PZ93] introduced α-*fairness*, which was shown to be sound and complete. The work there also introduced an explicit-state model checker for P-validity that is based on α-fairness. The main drawback of α-fairness is that it requires fairness with respect to *every past* formula. From the model checking procedure of [PZ93] it follows that the only past formulae really needed are those that appear in the normal-form of the property to be verified. However, obtaining the normal-form is non-elementary in the size of the property.

In [ZPK02] we observed that a consequence of the model checking procedure of [PZ93] is that replacing α-fairness by the simpler γ-fairness results in a notion of fairness that is sound and complete for proving *simple* properties over finite-state programs. The definition of γ fairness is as follows:

Assume a PDS $S : \langle V, \Theta, \rho, \mathcal{P}, \mathcal{J}, \mathcal{C} \rangle$ and a path $\pi = s_0, s_1, \ldots$ in a computation tree of S. Let (r, t_1, t_2) be a probabilistic selection. We say that mode (r, t_j), $j \in \{1, 2\}$ is *taken* at position $i \geq 0$ of π if $s_i \models r$ and $s_{i+1} \models t_j$. We say that the selection (r, t_1, t_2) is taken at position i if either (r, t_1) or (r, t_2) is taken at i. Let s be a state of the system. We call i an s-position if $s_i = s$. We say that π is γ-*fair*, if for each state s (and there are only finitely many distinct states) and each probabilistic selection (r, t_1, t_2), either there are only finitely s-positions from which (r, t_1, t_2) is taken, or for every $i = 1, 2$, there are infinitely many s-positions from which (r, t_i) is taken. The following corollary states that the replacement of probability by γ-fairness is sound and complete with respect to P-validity of simple formulae. It is an immediate consequence of [PZ86a].

Corollary 1. *For every finite-state* PDS S *and simple formula* φ*,* φ *is P-valid over* S *iff* $\sigma \models \varphi$ *for every* γ*-fair computation*

Based on Corollary 1, the explicit-state model checking procedure of [PZ93], and the non-probabilistic feasibility algorithm of [KPR98], we introduce SYMPMC, a symbolic model checker for verifying P-validity of simple properties over finite-state PDSs.

The core of SYMPMC is an algorithm for simple response formulae (formulae of the form $\Box(a \rightarrow \diamondsuit b)$, where a and b are assertions) over a finite state PDS. This algorithm is presented in Fig. 2. It checks the P-validity of $\varphi : \Box(a \rightarrow \diamondsuit b)$ for assertions a and b. The algorithm returns $\emptyset$ iff φ is γ-valid over S, i.e., if φ holds over all γ-fair computations of S. SYMPMC had been implemented using TLV [PS96].

```
ALGORITHM RESPONSE(S)
var:
  R:     relation init |ρ| ∩ (‖¬b‖ × ‖¬b‖)
  new:   predicate init (‖Θ‖ ∘ ‖ρ‖*) ∩ ‖¬b‖
  oldR:  relation init ∅
  old:   predicate init ∅

while (new ≠ old ∨ R ≠ oldR) do
   old := new
   oldR := R
   new := new ∩ (R ∘ new)
   R := R ∩ (new × new)
   foreach J ∈ 𝒥 do
         new :=  new ∩ R*∘ ‖J‖
         R:= R ∩ (new × new)
   foreach (p, q) ∈ 𝒞 do
         new := (new - ‖p‖) ∪
               (new ∩ R*∘ ‖q‖)
         R := R ∩ (new × new)
   foreach R ∈ 𝒫 do
         treat-P-requirement(R)
endwhile
return
   (‖Θ‖ ∘ ‖ρ‖*) ∩ (‖a‖ − ‖b‖) ∩ (R*∘ new)
```

where for a probabilistic requirement $R = (r, t_1, t_2)$,

```
treat-P-req(R):
  var:
    qpred: predicate init Σ
    someq: predicate init ∅
    pbad:  predicate

  for j =1 to 2 do
    qpred := qpred ∩ (R ∘ ‖t_j‖)
    someq := someq ∪ ‖t_j‖
    pbad := ‖r‖ - qpred
    R :=  R ∩
          [(pbad × (Σ - someq))
            ∪ ((Σ - pbad)×Σ)]
```

Fig. 2. Algorithm RESPONSE **for model-checking the P-validity of** $\varphi : \Box(a \rightarrow \diamondsuit b)$

The main difference between the algorithm in Fig. 2 and its counterpart in [KPR98] is the treatment of probabilistic requirements (the third "`foreach`" in the `while` loop). For each probabilistic requirement $R = (r, t_1, t_2)$, the procedure `treat-P-req(R)` removes from the graph all states that are not γ-fair with respect to R.

In [APZ03] we prove:

Theorem 1. *For an input* PDS S*, Algorithm* RESPONSE *terminates. For assertions* a *and* b*,* RESPONSE *returns* V *such that* $\varphi = \Box(a \rightarrow \diamondsuit b)$ *is P-valid in* S *iff* $V = \emptyset$.

By setting a and b to true, Algorithm RESPONSE can be used to check for γ*-feasibility* (whether the system has at least one γ-fair computation). It can also be used to check the validity of simple formulae by composing the system with temporal testers [KPR98]. Thus, SYMPMC can be used for symbolic model checking of whether a simple temporal formula is P-valid over a finite state program.

4 Probabilistic Parameterized Systems

In this section we turn to probabilistic parameterized systems and their automatic verification. We first define the systems that are the scope of this paper and briefly discuss their automatic verification using Planners and SYMPMC.

4.1 Parameterized Systems

We focus on probabilistic parameterized systems that consist of multiple copies of N identical finite-state SPL processes. For each value of $N > 0$, $S(N)$, the PDS that describes the system, is an instantiation of an PDS. Thus, such a system represents an infinite *family* of systems, one for each value of N.

We are interested in properties that hold for every process in the system. Thus, we are interested in liveness properties of the type φ, where φ is a temporal formula referring only to variables local to a single process. The problem of parameterized verification is to show that φ is P-valid over *every* $S(N)$.

4.2 Counter Abstraction

In [PXZ02] we proposed the method of *counter abstraction* for the verification of liveness properties of parameterized systems. A brief overview of the approach for non-probabilistic systems is given here. For details see [PXZ02].

For simplicity of presentation, we assume that the system $S(N)$ has a set X of global shared variables whose size is independent of N, and the only variable local to each process $P[i]$ is the program counter $\pi[i]$. Each global state s of the system $S(N)$ is then an $(N + |X|)$-tuple, describing the location of each process and the values of each $x \in X$. Assume that the program counters range over the set $\{0 \ldots L-1\}$.

We define the *counter abstraction* of state s by an $(L + |X|)$-tuple, such that each one of the first $|L|$ elements is the *counter* of the corresponding location, where for a location ℓ, the counter of ℓ, denoted by κ_ℓ, is defined by:

$$\kappa_\ell \;=\; \begin{cases} 0 & - \quad \text{there are no processes in location } \ell \\ 1 & - \quad \text{there is exactly one process in location } \ell \\ 2 & - \quad \text{there are two or more processes in location } \ell \end{cases}$$

Properties are similarly abstracted. Thus, for example, the property $\exists i : at\_\ell_\ell[i]$ is abstracted to $\kappa_\ell > 0$. Denote by $\alpha(\varphi)$ the counter-abstraction of the property φ.

As explained in [PXZ02], in order to be able to prove liveness properties it is necessary to carefully abstract the fairness properties. Once this is done, we obtain the abstracted system $\alpha(S)$, and can show that for every liveness property φ, the validity of $\alpha(\varphi)$ over $\alpha(S)$ implies the validity of φ over $S(N)$ for every $N > 1$.

Suppose we have a probabilistic parameterized system $S(N)$ and a temporal property φ we wish to show is P-valid. We can apply a Planner transformation, obtaining a non-probabilistic parameterized system, to which we can then apply counter-abstraction, which will reduce it to an unparameterized finite-state system. If model

checking reveals that $\alpha(\varphi)$ is valid for this system, we can safely conclude that φ is P-valid over $S(N)$.

It is important to note that counter-abstraction can be obtained in a fully automatic way. Obviously, model checking techniques can easily check whether an abstracted system satisfies abstract properties. The only step in the process that requires user intervention (and ingenuity) is in crafting the functions used by the Planner.

In Section 5.1 we demonstrate the power of the approach by verifying the liveness of a probabilistic N-process mutual exclusion algorithm. Previous attempts to verify the same protocol were either manual [PZ86a] or automatic for $N \leq 10$ [KNP00].

4.3 Network Invariants

The method of network invariants was first mentioned in [BCG86, SG89], further developed in [WL89] (who also coined the name "network invariant"), and elaborated in [KM95] into a working method. The formulation here follows [KP00] and [KPSZ02], which take into account the fairness properties of the compared systems and support proofs of liveness properties.

In order to apply the method to PDSs, it is necessary to refine the model, so that it allows for "environment" actions. Roughly speaking, the set of variables includes a special subset consisting of the variables *owned* by the system. The system then takes steps, alternating between environment steps that can change all but the owned variables.

It is also necessary to define the *observable behavior* of a system. To that end, the set of variables is assumed to includes a set of *observables*—variables that are externally observable. The observables are denoted by $\mathcal{O}$.

A γ-*observation* of S is a projection of a γ-fair computation of S onto $\mathcal{O}$. We denote by $Obs_\gamma(S)$ the set of all γ-observations of S. Systems $\mathcal{S}_C$ and $\mathcal{S}_A$ are said to be *comparable* if they have the same sets of observable variables,. System $\mathcal{S}_A$, is said to be a γ-*abstraction* of the comparable system $\mathcal{S}_C$, denoted $\mathcal{S}_C \sqsubseteq_\gamma \mathcal{S}_A$, if $Obs_\gamma(\mathcal{S}_C) \subseteq Obs_\gamma(\mathcal{S}_A)$. The abstraction relation is reflexive, transitive, and compositional, that is, whenever $\mathcal{S}_C \sqsubseteq_\gamma \mathcal{S}_A$ then $(\mathcal{S}_C \| Q) \sqsubseteq_\gamma (\mathcal{S}_A \| Q)$. It is also *property restricting*, that is, if $\mathcal{S}_C \sqsubseteq_\gamma \mathcal{S}_A$ then $\mathcal{S}_A \models_\gamma p$ implies that $\mathcal{S}_C \models_\gamma p$.

Suppose we are given two comparable systems, a *concrete* $\mathcal{D}_C$ and an *abstract* $\mathcal{D}_A$, and wish to establish that $\mathcal{D}_C \sqsubseteq_\gamma \mathcal{D}_A$. Without loss of generality, we assume that $V_C \cap V_A = \emptyset$, and that there exists a 1-1 correspondence between the concrete observables $\mathcal{O}_C$ and the abstract observables $\mathcal{O}_A$.

In Fig. 3, we present a rule for proving that S_A γ-abstracts S_C. The rule assumes the identification of an *abstraction mapping* $\alpha : (U = \mathcal{E}_\alpha(V_C))$ which assigns expressions over the concrete variables to *some* of the abstract variables $U \subseteq V_A$. For an abstract assertion φ, we denote by $\varphi[\alpha]$ the assertion obtained by replacing the variables in U by their concrete expressions.

The Abstraction Rule resembles the abstraction rule of [KPSZ02], with the addition of Premise A6. This premise must, in general, be verified for every assignment U to the abstract variables V_A. The following condition suffices to guarantee premise A6 and is met in many abstractions:

> For every abstract requirement $(r, t_1, t_2) \in \mathcal{P}_A$, there exists a concrete requirement $(r^C, t_1^C, t_2^C) \in \mathcal{P}_C$, where $r[\alpha] = r^C$, $t_1[\alpha] = t_1^C$ and $t_2[\alpha] = t_2^C$.

$$
\begin{array}{ll}
\mathbf{A1}.\ \Theta_C \rightarrow \exists V_A : \Theta_A[\alpha] & \\
\mathbf{A2}.\ \mathcal{D}_C \models \Box(\rho_C \rightarrow \exists V'_A : \rho_A[\alpha][\alpha']) & \\
\mathbf{A3}.\ \mathcal{D}_C \models \Box(\alpha \rightarrow \mathcal{O}_C = \mathcal{O}_A) & \\
\mathbf{A4}.\ \mathcal{D}_C \models \Box \Diamond J[\alpha], & \text{for every } J \in \mathcal{J}_A \\
\mathbf{A5}.\ \mathcal{D}_C \models \Box \Diamond p[\alpha] \rightarrow \Box \Diamond q[\alpha], & \text{for every } (p,q) \in \mathcal{C}_A \\
\mathbf{A6}.\ \mathcal{D}_C \models \Box \Diamond((V_A = U)[\alpha] \wedge r[\alpha] \wedge \bigcirc \bigvee_{i=1}^{2} t_i[\alpha]) \rightarrow & \\
\qquad \bigwedge_{i=1}^{2} \Box \Diamond((V_A = U)[\alpha] \wedge r[\alpha] \wedge \bigcirc t_i[\alpha]), & \\
\qquad \text{for every } (r, t_1, t_1) \in \mathcal{P}_A \text{ and every assignment } U \text{ over } V_A & \\
\hline
S_C \sqsubseteq_\gamma S_A &
\end{array}
$$

Fig. 3. Abstraction Rule

Given a parameterized system $S(N)$, the network invariant method calls for devising a *network invariant* $\mathcal{I}$ — a finite state PDS, intended to provide an abstraction for the (open) parallel composition of any k processes of the parameterized system. The method then calls for confirming that $\mathcal{I}$ is indeed an abstraction, and model checking that when composed with a single process it satisfies a property of that process. The first step, that of designing $\mathcal{I}$, calls for some ingenuity of the verifier. As we showed in [KPSZ02], the task can be often quite simple. The third step can be achieved using SYMPMC. The second step, confirming that $\mathcal{I}$ is indeed a good abstraction, calls for establishing the two γ-abstractions $(P[1] \| \cdots \| P[m]) \sqsubseteq_\gamma \mathcal{I}$ and $(\mathcal{I} \| P[i]) \sqsubseteq_\gamma \mathcal{I}$, where m is a small constant (usually in the range $[1..3]$) independent of N, and $P[i]$ is a generic copy of the system's process.

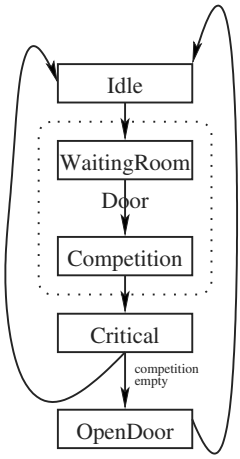
5 Examples

In this section we present two examples, one for each of the methodologies we described in the previous section. To demonstrate the power of the "Planner and Counter abstraction" approach, we take the probabilistic mutual exclusion protocol of [PZ86b]. To demonstrate the power of the "SYMPMC and network invariant" approach, we take the Lehman and Rabin's Courteous Philosopher protocol [LR81].

5.1 Verifying Probabilistic Mutual Exclusion Using a Planner

A flow diagram of the protocol, as well as its SPL code, are in Fig. 4. The usual "trying section" consists of two parts: A "waiting room" in which a process waits for a "door" to open in order to be admitted to the "competition", and the competition. Once the door closes, no new process can enter the competition. Processes in the competition flip coins: Losers, those who flip *tails*, wait until there are no winners. A process that flips *heads*, and finds out that it is the only one to have done so (and there are no processes in the critical or exit region), enters the critical section. Otherwise, it waits until all winners join it, and they all proceed to flip coins again. Once a process leaves the critical region, it examines if there are processes in the competition. If there are, it just goes back to its idle state. Otherwise, it opens the door and waits until all processes in the waiting room enter the competition before it goes back to the idle state.

In the original protocol each process has a variable y that can take on eight values, according to the location(s) of the process. We omit it here, and instead present a version that includes locations only, and is amenable to counter-abstraction. Each process i can perform, in a single atomic step, a test consisting of a boolean combination of formulae of the form $\exists j \neq i : \pi[j] \in L$ for $L \subseteq [0..15]$. We denote such a test by *some* $\in L$, and its negation by *none* $\in L$. The only probabilistic transition is at location ℓ_4.



in n : **integer where** $n \geq 2$

$$\|_{i=1}^{n} P[i] ::$$

loop forever do

ℓ_0 : **non-critical**
ℓ_1 : **skip**
ℓ_2 : **await** *none* $\in \{4..13\} \ \vee$ *some* $\in \{14, 15\}$
ℓ_3 : **skip**
ℓ_4 : **pr_goto** $\{\ell_5, \ell_8\}$
ℓ_5 : **if** *none* $\in \{5, 6, 10..15\}$ **then goto** ℓ_{10}
ℓ_6 : **skip**
ℓ_7 : **await** *none* $\in \{5, 6, 10..15\}$
and then goto ℓ_4
ℓ_8 : **await** *none* $\in \{4..7, 10..15\}$
ℓ_9 : **goto** ℓ_4
ℓ_{10} : **Critical**
ℓ_{11} : **if** *none* $\in \{4..9\}$ **goto** ℓ_{13}
ℓ_{12} : **goto** ℓ_0
ℓ_{13} : **skip**
ℓ_{14} : **await** *none* $\in \{2, 3\}$
ℓ_{15} : **goto** ℓ_0

Fig. 4. A Probabilistic Mutual Exclusion Protocol

The mutual exclusion property of the protocol, stating that it is never the case that two or more processes are in ℓ_{10} at the same time, is easy to model check, for example using the methodology of [APR$^+$01] or counter-abstraction. The liveness property of the protocol is $\forall i : \pi[i] = 1 \rightarrow \diamondsuit(\pi[i] = 10)$, and we wish to show its P-validity for every $N \geq 2$.

We take $k = 1$ and define a planner *planner*$_4$ which determines the result of the next probabilistic choice at ℓ_4 whenever activated. This planner is defined by

planner$_4$: *none* $\in \{5, 6, 10..15\}$

This planner directs the next branch to ℓ_5 if there is no process in any of the locations $H = \{5, 6, 10..15\}$, and to ℓ_6 otherwise.

The intuition behind this planner is that a process can enter the critical section, from location ℓ_5, only if there are no other processes in H-locations. When there are pro-

cesses in H-locations, we want to reduce the number of processes that are likely to join them, which *planner*$_4$ accomplishes by returning "0" and thus forcing processes from ℓ_4 to enter ℓ_8. When there are no processes in H-locations, we want the first process that can to enter ℓ_5, so that it can enter the critical section. Thus, *planner*$_4$ returns "1" in such cases. This planner design can obviously be counter abstracted, hence, we succeeded to use TLV to establish the livelock-freedom property of the protocol given by: $\Box(\exists i : \pi[i] = 1 \rightarrow \Diamond(\exists i : \pi[i] = 10))$.

Once livelock freedom has been established, it is the structure of the protocol that guarantees individual liveness, by restricting the number of times a process in the competition can overtake another — once a process i is trying to access the critical section (enters $\ell_{2..9}$), every other process can enter the critical section at most twice before i does, which trivially implies the individual accessibility or the protocol [3].

We also established the individual liveness property of the protocol directly using TLV and "counter abstraction save one" ([PXZ02]) using the same planner. See *http://www.cs.nyu.edu/zuck/pubs/pme* for TLV code.

5.2 Verifying the Courteous Philosophers Using SYMPMC

The success of the planner strategy in parameterized systems depends on having a *single* strategy for random draws that will allow *every* process to achieve its liveness property. The [LR81] Courteous Philosophers Algorithm is an example where we cannot use a planner: any planner strategy that allows one philosopher to eat may preclude its neighbours from eating. Hence, to automatically verify The [LR81] Courteous Philosophers Algorithm, we use the network invariant approach.

The network invariant we obtained is essentially the one derived in [KPSZ02] and we omit it here for space reasons. There is, however, a crucial difference: In [KPSZ02] we replaced the probabilistic requirements by compassion requirements. Here, with the aid of the revised Abstraction Rule and SYMPMC, we could work directly with γ-fairness and did not need to (manually) devise adequate compassion requirements replacing the probabilistic choices. Thus, the resulting network invariant is significantly simpler and execution time is much shorter.

6 Conclusion and Future Work

The paper deals with the problem of automatic proof of P-validity of liveness properties over parameterized systems. We started with a discussion of the non-parameterized case, and described two new approaches to the problem: Planners that convert a probabilistic system into a non-probabilistic one and allow one to treat P-validity as regular validity, and model checking over γ-fair computations, which is sound and complete for simple temporal properties. We then outlined the two approaches of automatic verification of liveness properties of parameterized systems, counter abstraction and network invariants, and showed how the network invariant method can be combined with

[3] Bounded overtaking property is a safety property and thus it can be established by ignoring the probabilistic transitions of the protocol.

SYMPMC. We demonstrated our techniques by providing automatic proofs for two non-trivial protocols. The first by Planner & counter-abstraction, the second by SYMPMC & (extended) network invariants.

Strictly speaking, neither method combination we used in our examples is *fully automatic*, they both require user input. On one hand the design of a Planner or a Network Invariant may require user ingenuity; on the other hand, most systems are verified by their own designers, who have a pretty good intuition about the appropriate Planner/network invariant.

We are currently working on extending counter-abstraction with γ-fairness. If successful, this will provide a fully automatic proofs of P-validity of parameterized system for the cases that the method of counter-abstraction is applicable.

References

[AK86] K. R. Apt and D. Kozen. Limits for automatic program verification of finite-state concurrent systems. *Information Processing Letters*, 22(6), 1986.

[APR+01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *Proc.* 13^{th} *Intl. Conference on Computer Aided Verification (CAV'01), volume 2102 of*Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 221–234, 2001.

[APZ03] T. Arons, A. Pnueli, and L. Zuck. Verification by probabilistic abstraction. Weizmann Institute of Science Technical Report, 2003.

[BCG86] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many finite state processes. In *Proc. 5th ACM Symp. Princ. of Dist. Comp.*, pages 240–248, 1986.

[BLS01] K. Baukus, Y. Lakhnesche, and K. Stahl. Verification of parameterized protocols. Journal of Universal Computer Science, 7(2):141–158, 2001.

[CLP84] S. Cohen, D. Lehmann, and A. Pnueli. Symmetric and economical solutions to the mutual exclusion problem in a distributed system. *Theor. Comp. Sci.*, 34:215–225, 1984.

[EK00] E.A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *17th International Conference on Automated Deduction (CADE-17)*, pages 236–255, 2000.

[EN95] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. 22th ACM Conf. on Principles of Programming Languages, POPL'95*, San Francisco, 1995.

[EN96] E.A. Emerson and K.S. Namjoshi. Automatic verification of parameterized synchronous systems. In *R. Alur and T. Henzinger, editors,* Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96)*, volume 1102 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, 1996.

[Fel68] W. Feller. *An Introduction to Probability Theory and its Applicaitons*, volume 1. John Wiley & Sons, 3 edition, 1968.

[HSP82] S. Hart, M. Sharir, and A. Pnueli. Termination of probabilistic concurrent programs. In *Proc. 9th ACM Symp. Princ. of Prog. Lang.*, pages 1–6, 1982.

[KM95] R.P. Kurshan and K.L. McMillan. A structural induction theorem for processes. *Information and Computation*, 117:1–11, 1995.

[KNP00] M. Kwiatkowska, G. Norman, and D. Parker. Verifying randomized distributed algorithms with prism. In *Proc. of the Workshop on Advances in Verification (WAVe) 2000*. 2000.

[KNP02] M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. In TOOLS 2002, volume 2324 of LNCS, 2002.

[KP00] Y. Kesten and A. Pnueli. Control and data abstractions: The cornerstones of practical formal verification. *Software Tools for Technology Transfer*, 4(2):328–342, 2000.

[KPR98] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proc. 25th Int. Colloq. Aut. Lang. Prog.*, volume 1443 of *LNCS*, pages 1–16. Springer-Verlag, 1998.

[KPSZ02] Y. Kesten, A. Pnueli, E. Shahar, and L. Zuck. Network invariants in action. In *Proceedings of Concur'02*, volume 2421 of *LNCS*. Springer-Verlag, 2002.

[LR81] D. Lehmann and M.O. Rabin. On the advantages of free choice: A symmetric and fully distibuted solution to the dining philosophers problem (exended abstract). In *Proc. 8th ACM Symp. Princ. of Prog. Lang.*, pages 133–138, 1981.

[Lub84] B.D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs. Acta Infromatica, 21, 1984.

[MAB+94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, Stanford, California, 1994.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[Pnu83] A. Pnueli. On the extremely fair treatment of probabilistic algorithms. In *Proc. 15th ACM Symp. Theory of Comp.*, pages 278–290, 1983.

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In *R. Alur and T. Henzinger, editors,* Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96)*, volume 1102 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 184–195, 1996.

[PXZ02] A. Pnueli, J. Xu, and L. Zuck. The $(0, 1, \infty)$ counter abstraction. In *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02), volume 2404 of*Lect. Notes in Comp. Sci.*, Springer-Verlag*, 2002. *http://www.cs.nyu.edu/~zuck/pubs/cav02.ps*.

[PZ86a] A. Pnueli and L. Zuck. Probablistic verification by tableaux. In *Proc. First IEEE Symp. Logic in Comp. Sci.*, pages 322–331, 1986.

[PZ86b] A. Pnueli and L. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1:53–72, 1986.

[PZ93] A. Pnueli and L.D. Zuck. Probabilistic verification. *Information and Computation*, 103(1):1–29, 1993.

[Rab82] M.O. Rabin. The choice coordination problem. *Acta Informatica*, 17:121–134, 1982.

[SG89] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 151–165. Springer-Verlag, 1989.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symp. Logic in Comp. Sci.*, pages 332–344, 1986.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 68–80. Springer-Verlag, 1989.

[ZPK02] L. Zuck, A. Pnueli, and Y. Kesten. Automatic verification of probabilistic free choice. In *Proc. of the* 3^{rd} *workshop on Verification, Model Checking, and Abstract Interpretation*, volume 2294 of *LNCS*, 2002.

A Soundness of the Planner Approach

We prove the soundness of the planner method by reducing its soundness to that of α-fairness, which was established in [PZ86a].

Let $Q : \langle V, \Theta, \rho, \mathcal{P}, \mathcal{J}, \mathcal{C} \rangle$ be a PDS. We make two simplifying assumptions: (1) $\mathcal{P}$ contains exactly one requirement, $R = (r, t_1, t_2)$, and (2) there is only one location from which R is taken. The proof below is easy to generalize once the two assumptions are removed, at the cost of additional indices.

Let $\sigma = s_0, \ldots$ be a fair computation of Q and let ψ be a past temporal formula, that is, a formula whose only temporal operators are past operators. Computation σ is α-fair with respect to ψ if either R is taken only finitely many times from ψ-prefixes in σ, or each mode of R is taken infinitely many times from ψ-prefixes in σ. For a set $\mathcal{X}$ of temporal formula, σ is $\mathcal{X}$-fair if it is α-fair with respect to every $\psi \in \mathcal{X}$.

An immediate corollary of the analysis presented in [PZ86a] is:

Corollary 2. . *Let $\mathcal{X}$ be a set of past formulae and φ be a temporal formula. Then:*

$$\sigma \models \varphi \text{ for every } \mathcal{X}\text{-fair } \sigma \quad \Longrightarrow \quad \varphi \text{ is P-valid over } Q.$$

Consider now the PDS Q_T obtained from Q by a transforming it with a Planner *planner*, which is set to make k consecutive "planned" choices. We proceed to construct a set of past formulae $\mathcal{X}$ such that every $\mathcal{X}$-fair computation of Q is a V-projection of some computation of $Q_{_T}$. Thus, for every temporal formula φ over V, if all $Q_{_T}$ computations satisfy φ, then all $\mathcal{X}$-fair computations of Q also satisfy φ. It then follows from Corollary 2 that φ is P-valid. It thus remains to construct $\mathcal{X}$ and show that every $\mathcal{X}$-fair computation of Q has a computation of $Q_{_T}$ with the same V-projection.

For $i = 1, 2$, let p_i be the state assertion characterizing the r-states for which *planner* recommends choosing i. Define $chose : (t_1 \vee t_2) \wedge \ominus r$. Formula $chose$ characterizes all the states immediately following a probabilistic choice. Next, define $gc : \bigvee_{i=1,2}(t_i \wedge \ominus(r \wedge p_i))$. Formula gc (standing for *good-choice*) characterizes all the states following a probabilistic choice which is compatible with the choice recommended by the planner at this state. Finally, define

$$\begin{array}{ll} \psi_0 & : \mathrm{T} \\ \psi_{i+1} & : (\neg chose)\, \mathcal{S}\, (gc \ \wedge \ \ominus \psi_i) \qquad\qquad \text{for } i = 0, \ldots, k-1 \end{array}$$

Obviously, ψ_i characterizes a point in the computation such that the last i probabilistic choices are compatible with the recommendations of the planner. As the set of past formulae, we take $\mathcal{X} : \{\psi_0, \ldots, \psi_k\}$.

It remains to show that every $\mathcal{X}$-fair computation of Q has a computation of $Q_{_T}$ with the same V-projection. Let σ be a $\mathcal{X}$-fair computation of Q, and assume that σ has infinitely many R-transitions. By induction on $i = 1, \ldots, k$, (which is similar to the proof of the Borel-Cantelli Lemma) it follows that infinitely many times, σ makes k consecutive choices which are compatible with the recommendation of the planner. It is easy now to construct a computation of $Q_{_T}$ in which the planner is activated whenever σ is to start a sequences of good choices, and is activated nowhere else.

The soundness of the Planner strategy follows.

Genericity and the π-Calculus

Martin Berger[1], Kohei Honda[1], and Nobuko Yoshida[2]

[1] Department of Computer Science, Queen Mary, University of London, London, U.K.
[2] Department of Computing, Imperial College, London, U.K.

Abstract. We introduce a second-order polymorphic π-calculus based on duality principles. The calculus and its behavioural theories cleanly capture some of the core elements of significant technical development on polymorphic calculi in the past. This allows precise embedding of generic sequential functions as well as seamless integration with imperative constructs such as state and concurrency. Two behavioural theories are presented and studied, one based on a second-order logical relation and the other based on a polymorphic labelled transition system. The former gives a sound and complete characterisation of the contextual congruence, while the latter offers a tractable reasoning tool for a wide range of generic behaviours. The applicability of these theories is demonstrated through non-trivial reasoning examples and a fully abstract embedding of System F, the second-order polymorphic λ-calculus.

1 Introduction

Genericity is a useful concept in software engineering which allows encapsulation of design decisions such that data-structures and algorithms can be changed more independently. It arises in two distinct but closely related forms: one, which we may refer to as *universal*, aids generic manipulation of data, as in lists, queues, trees or stacks. The other *existential* form facilitates hiding of structure from the outside, asking for it to be treated generically. In both cases, genericity partitions programs into parts that depend on the precise nature of the data under manipulation and parts that do not, supporting principled code reuse and precise type-checking. For example, C++ evolved from C by adding genericity in the form of *templates* (universal) and *objects* (existential).

It is known that key aspects of genericity for sequential functional computation are captured by second-order polymorphism where type variables, in addition to program variables, can be abstracted and instantiated. In particular, the two forms of genericity mentioned above are accounted for by the two forms of quantification coming from logic, ∀ and ∃. Basic formalisms incorporating genericity include System F (the second-order λ-calculus) [8, 28] and ML [18]. Centring on these and related formalisms, a rich body of studies on type disciplines, semantics and proof principles for genericity has been accumulated.

The present work aims to offer a π-calculus based starting point for repositioning and generalising the preceding functional account of genericity in the

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 103–119, 2003.

broader realm of interaction. We are partly motivated by the lack of a general mathematical basis of genericity that also covers state, concurrency and nondeterminism. For example, the status of two fundamental concepts for reasoning about generic computation, relational parametricity [28] and its dual simulation principle [1, 19, 27], is only well-understood for pure functions. But a mathematical basis of diverse forms of generic computation is important when we wish to reason about software made up from many components with distinct behavioural properties, from purely functional behaviour to programs with side effects to distributed computing, all of which may exhibit certain forms of genericity.

The π-calculus is a small syntax for communicating processes in which we can precisely represent many classes of computational behaviours, from purely sequential functions to those of distributed systems [5, 7, 17, 32, 33]. Can we find a uniform account of genericity for diverse classes of computational behaviour using the π-calculus? This work presents our initial results in this direction, concentrating on a polymorphic variant of the linear/affine π-calculus with state [7, 12, 32, 33]. It turns out that the duality principle in the linear/affine type structure naturally extends to second-order quantification, leading to a powerful theory of polymorphism that allows precise embedding of existing polymorphic functional calculi and unifies some of the significant technical elements of the known theories of genericity.

Summary of Contributions. The following summarises the main technical contributions of the present paper.

1. Introduction of the polymorphic linear/affine π-calculus based on duality principles, as well as its consistent extension to state and concurrency. One of the central syntactic results is strong normalisability for linear polymorphic processes.
2. Theory of behavioural equivalences based on a generic labelled transition system applicable to both sequential and concurrent polymorphic processes. We apply the theory to non-trivial reasoning examples as well as to a fully abstract embedding of System F in the linear polymorphic π-calculus.
3. A sound and complete characterisation of the contextual congruence by a second-order logical relation for linear/affine polymorphic processes, leading to relational parametricity and a simulation principle for extensional equality. The theory offers a tractable reasoning tool for generic processes as we demonstrate through examples.

Related Work. Originally the second-order polymorphism for the λ-calculus was discovered by Girard [8] and Reynolds [28] with a main focus on universal abstraction. Later Mitchell and Plotkin [19, 27] relates its dual form, existential abstraction, to data hiding. Exploiting a duality principle, the present theory unifies these two uses of polymorphism, data-hiding and parametricity, into a single framework, both in operation and in typing. The unification accompanies new reasoning techniques such as generic labelled transition.

Turner [30] is the first to study (impredicative) polymorphism in the π-calculus, giving a type-preserving encoding of System F. His type discipline is incorporated into Pict [23]. Vasconcelos [31] studies a predicative polymorphic typing discipline and shows that it can type-check interesting polymorphic processes while allowing tractable type inference. Our use of a duality principle (whose origin can be traced back to Linear Logic [9]) is the main difference from those previous approaches.

Pierce and Sangiorgi [22] study a behavioural equivalence for Turner's calculus and observe that existential types can reduce the number of transitions by prohibiting interactions at hidden channels. Lazic, Nowak and Roscoe [15] show that when programs manipulate data abstractly (called *data independence*), a transition system with a parametricity property can be used for reasoning, leading to efficient model checking techniques. The generic labelled transition unifies, and in some cases strengthens, these ideas as dual aspects of a single framework. The use of duality also leads to lean and simple constructions.

Pitts studies contextual congruences in PCF-like polymorphic functional calculi and characterises them via syntactic logical relations [25, 26], cf. [24]. His work has inspired constructions and proof techniques for our corresponding characterisations. The present relational theory for the π-calculus treats several elements of Pitts' theories (for example call-by-name and call-by-value) in a uniform framework. Duality also substantially simplifies the constructions.

Recently, several studies of the semantics of polymorphism based on games and other intensional models have appeared. Hughes [13] presents game semantics for polymorphism in which strategies pass arenas to represent type passing and proves full abstraction for System F. His model is somewhat complex due to its direct representation of type instantiation. Murawski and Ong [21] substantially simplify Hughes approach, but do not obtain full abstraction for impredicative polymorphism. Abramsky and Lenisa [3, 4] give a fully abstract model for predicative polymorphism using interaction combinators. Treatment of impredicative polymorphism is left as an open issue. In view of the relationship between π-calculi and game semantics [7, 11, 14], it would be interesting to use typed processes from the present work to construct game-based categories.

Structure of the paper. Section 2 informally illustrates key ideas with examples. Section 3 introduces the syntax and typing rules. Section 4 gives a sound and complete characterisation of a contextual congruence by a second-order logical relation. Section 5 studies a generic labelled transition and the induced equivalence. Section 6 discusses non-trivial applications of two behavioural theories, including a fully abstract embedding of System F. The full technical development of the presented material is found in [6].

Acknowledgement. We thank anonymous referees for their helpful suggestions. The first two authors are partially supported by EPSRC grant GR/N/37633. The third author is partially supported by EPSRC grant GR/R33465/01.

2 Generic Processes, Informally

This section introduces key ideas with simple examples. We start with the following small polymorphic process (which is essentially a process encoding of the polymorphic identity), using the standard syntax of the (asynchronous) π-calculus.

$$\vdash !x(yz).\overline{z}\langle y\rangle \triangleright x : \forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$$

In this process $\overline{z}\langle y\rangle$ is an output of y along the channel z and $!x(yz).\overline{z}\langle y\rangle$ is a replicated input, repeatedly receiving two names y and z at x. After having received y and z, it sends y along z.

The typing $x : \forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$ assigns $\forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$ to x. X is a type variable: $\overline{\mathrm{X}}$ indicates the dual of X. $(\mathrm{X})^{\uparrow}$ sends a name of type X exactly once, while $(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$ indicates the behaviour of receiving two names at a replicated input channel, one used as $\overline{\mathrm{X}}$ and the other as $(\mathrm{X})^{\uparrow}$. Finally, $\forall \mathrm{X}$ universally abstracts X, saying X can be any type. Here $\forall \mathrm{X}$ binds X and its dual simultaneously. The operational content of typing a channel with a type variable is to enforce that y cannot be used as an interaction point (which would require a concrete type). Hence y with a variable $\overline{\mathrm{X}}$ only appears as a value in a message.

Next we consider the process which is dual to the above agent. Let $\mathsf{t}\langle y\rangle \stackrel{\text{def}}{=} !y(a_1a_2z).\overline{z}\langle a_1\rangle$, $\mathsf{not}\langle cw\rangle \stackrel{\text{def}}{=} !c(a_1a_2z).\overline{w}\langle a_2a_1z\rangle$ and $\mathbb{B} \stackrel{\text{def}}{=} \forall \mathrm{X}.(\overline{\mathrm{X}}\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$ (which are, respectively, truth, negation and the polymorphic boolean type).

$$\vdash \overline{x}(yz)(\mathsf{t}\langle y\rangle | z(w).\overline{e}(c)\mathsf{not}\langle cw\rangle) \triangleright x : \exists \mathrm{X}.(\mathrm{X}(\overline{\mathrm{X}})^{\downarrow})^{?}, e : (\mathbb{B})^{\uparrow} \tag{1}$$

This process sends y and z (respectively representing the truth and the continuation) via x, where $\overline{x}(yz)P$ stands for $(\boldsymbol{\nu}\, yz)(\overline{x}\langle yz\rangle | P)$. Then it receives a single name at z and sends its negation via e. To understand the typing, let's look at the situation before existential abstraction:

$$\vdash \overline{x}(yz)(\mathsf{t}\langle y\rangle | z(w).\overline{e}(c)\mathsf{not}\langle cw\rangle) \triangleright x : (\mathbb{B}(\overline{\mathbb{B}})^{\downarrow})^{?}, e : (\mathbb{B})^{\uparrow} \tag{2}$$

We now abstract $\mathbb{B}$ and its dual at x simultaneously, obtaining $\exists \mathrm{X}.(\mathrm{X}(\overline{\mathrm{X}})^{\downarrow})^{?}$ ($\exists \mathrm{X}$ binds both X and $\overline{\mathrm{X}}$). Thus existential abstraction hides the concrete type $\mathbb{B}$.

The types $\forall \mathrm{X}.(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{!}$ and $\exists \mathrm{X}.(\mathrm{X}(\overline{\mathrm{X}})^{\downarrow})^{?}$ are dual to each other and indicate that composition of two processes is possible. When composed, the process interacts as follows. Below and henceforth we write $\mathsf{id}\langle x\rangle$ for $!x(yz).\overline{z}\langle y\rangle$.

$$\begin{array}{rl} \mathsf{id}\langle x\rangle \,|\, \overline{x}(yz)(\mathsf{t}\langle y\rangle | z(w).\overline{e}(c)\mathsf{not}\langle cw\rangle) & \longrightarrow \mathsf{id}\langle x\rangle \,|\, (\boldsymbol{\nu}\, yz)(\overline{z}\langle y\rangle | \mathsf{t}\langle y\rangle | z(w).\overline{e}(c)\mathsf{not}\langle cw\rangle) \\ & \longrightarrow \mathsf{id}\langle x\rangle \mid (\boldsymbol{\nu}\, y)(\mathsf{t}\langle y\rangle | \overline{e}(c)\mathsf{not}\langle cy\rangle) \\ & \approx \mathsf{id}\langle x\rangle \mid \overline{e}(c)\mathsf{f}\langle c\rangle \end{array}$$

Here $\mathsf{f}\langle c\rangle \stackrel{\text{def}}{=} !c(xyz).\overline{z}\langle y\rangle$ (representing falsity) and $\approx$ is the standard weak bisimilarity. As this interaction indicates, a universally abstracted name, after its receipt from the environment, can only be used to be sent back to the environment as a free name. The dual existential side can then count on such behaviour of the interacting party: in the above case, the process on the right-hand side can

expect that, via z, it would receive the name y as a free name which it has exported in the initial reduction, as it indeed does in the second transition.

This duality plays the key role in defining generic labelled transitions, which induce behavioural equivalences more abstract (larger) than non-generic ones and which are applicable to the reasoning over a wide range of generic behaviours. We use an example of a generic transition sequence of the process in (1).

$$\overline{x}(yz)(\mathsf{t}\langle y\rangle|z(w).\overline{e}(c)\mathsf{not}\langle cw\rangle) \stackrel{\overline{x}(yz)}{\longrightarrow} \stackrel{z\langle y\rangle}{\longrightarrow} \mathsf{t}\langle y\rangle|\overline{e}(c)\mathsf{not}\langle cy\rangle$$

A crucial point in this transition is that it does *not* allow a bound input in the second action, because the protocol at existentially abstracted names is opaque. The induced name substitution then opens a channel for internal communication. In contrast, the process in (2), different from (1) only in type, has the following transition sequence.

$$\overline{x}(yz)(\mathsf{t}\langle y\rangle|z(w).\overline{e}(c)\mathsf{not}\langle cw\rangle) \stackrel{\overline{x}(yz)}{\longrightarrow} \stackrel{z(w)}{\longrightarrow} \mathsf{t}\langle y\rangle|\overline{e}(c)\mathsf{not}\langle cw\rangle.$$

Note that we have a bound input in the second action; the transition sequence is now completely controlled by type information, without sending/receiving concrete values. Here the duality principle dictates existential/universal type variables correspond to free name passing, while concrete types (which rigorously specify protocols of interaction by their type structure) correspond to bound name passing.

This way, the duality in the type structure is precisely reflected in the duality in behaviour. This duality principle is also essential in the construction of the second-order logical relations, for proving the strong normalisability of linear polymorphic processes and for various embedding results.

3 A Polymorphic π-Calculus

3.1 Processes

In this section we formally introduce a polymorphic version of the affine π-calculus [7] and its extensions to linearity [32, 33] and state [12].

Let $x, y, \ldots$ range over a countable set $\mathcal{N}$ of *names*. $\vec{y}$ is a vector of names. Then *processes*, ranged over by $P, Q, R, \ldots$, are given by the following grammar.

$$P \;::=\; x(\vec{y}).P \mid !x(\vec{y}).P \mid \overline{x}\langle\vec{y}\rangle \mid P|Q \mid (\boldsymbol{\nu}\, x)P \mid \mathbf{0}$$

Names in round parenthesis act as binders, based on which we define the alpha equality $\equiv_\alpha$ in the standard way. We briefly illustrate each construct. $x(\vec{y}).P$ inputs via x with a continuation P. Its replicated counterpart is $!x(\vec{y}).P$. $\overline{x}\langle\vec{y}\rangle$ outputs $\vec{y}$ along x. In each of these agents, the initial free occurrence is *subject*, while each carried name in an input/output is *object*. The parallel composition of P and Q is $P|Q$ and $(\boldsymbol{\nu}\, x)P$ makes x private to P. $\mathbf{0}$ is the inaction, indicating the lack of behaviour. The structural equality $\equiv$ is standard [17] (without the

replication rule $!P \equiv P|!P$), which we omit. The reduction relation $\longrightarrow$ are generated from:

$$x(\vec{y}).P \,|\, \overline{x}\langle\vec{z}\rangle \;\longrightarrow\; P\{\vec{z}/\vec{y}\} \qquad\qquad !x(\vec{y}).P \,|\, \overline{x}\langle\vec{z}\rangle \;\longrightarrow\; !x(\vec{y}).P \,|\, P\{\vec{z}/\vec{y}\}$$

closing under parallel composition and hiding, taking processes modulo $\equiv$.

3.2 Types

Below we introduce polymorphic extensions of different classes of first-order type disciplines, starting from the affine sequential polymorphic typing [7]. Following [7, 32, 33] we use the set of *action modes*, which are:

$\downarrow$ Affine input (at most once), $\uparrow$ Affine output (at most once),
! Server at replicated input, **?** Client requests to **!**,

as well as $\updownarrow$, which indicates non-composability at affine channels. $\downarrow, \mathbf{!}$ are *input modes*, while $\uparrow, \mathbf{?}$ are *output modes*. Input/output modes are together called *directed modes*. $p, p', \ldots$ (resp. p_{I}, resp. p_{O}) denote directed (resp. input, resp. output) modes. We define $\overline{p}$, the *dual* of p, by: $\overline{\downarrow} = \uparrow$, $\overline{\mathbf{!}} = \mathbf{?}$ and $\overline{\overline{p}} = p$.

Let $\mathrm{x}, \mathrm{x}', \ldots$ range over a countable set of *type variables*. We fix a bijection $\overline{\mathrm{x}}$ which is self-inverse (i.e. $\overline{\overline{\mathrm{x}}} = \mathrm{x}$) and irreflexive (i.e. $\overline{\mathrm{x}} \neq \mathrm{x}$). Each x is assigned a directed action mode p, written x^p, so that the mode of $\overline{\mathrm{x}}$ is always dual to that of x. Channel types are given as follows:

$$\tau ::= \tau_{\mathtt{I}} \mid \tau_{\mathtt{O}} \mid \langle \tau_{\mathtt{I}}, \tau'_{\mathtt{O}} \rangle \quad \tau_{\mathtt{I}} ::= \mathrm{x}^{p_{\mathrm{I}}} \mid (\vec{\tau_{\mathtt{O}}})^{p_{\mathrm{I}}} \mid \forall \mathrm{x}.\tau_{\mathtt{I}} \mid \exists \mathrm{x}.\tau_{\mathtt{I}} \quad \tau_{\mathtt{O}} ::= \mathrm{x}^{p_{\mathrm{O}}} \mid (\vec{\tau_{\mathtt{I}}})^{p_{\mathrm{O}}} \mid \forall \mathrm{x}.\tau_{\mathtt{O}} \mid \exists \mathrm{x}.\tau_{\mathtt{O}}$$

$\tau_{\mathtt{I}}$ and $\tau_{\mathtt{O}}$ are called *input type* and *output type*, respectively, which are together called *directed types*. Note quantification is given only on directed types. For each directed τ, the *dual of* τ, $\overline{\tau}$, is the result of dualising all action modes, type variables and quantifiers in τ. In $\langle \tau, \tau' \rangle$, we always assume $\tau' = \overline{\tau}$. Following [7, 12, 32, 33], we assume the sequentiality constraint on channel types, i.e. $\downarrow$-type carries only **?**-types while a **!**-type carries **?**-types and a unique $\uparrow$-type, dually for $\uparrow$ /**?**. We set $\mathsf{md}(\mathrm{x}^p) = p$, $\mathsf{md}((\vec{\tau})^p) = p$ and $\mathsf{md}(\forall \mathrm{x}.\tau) = \mathsf{md}(\exists \mathrm{x}.\tau) = \mathsf{md}(\tau)$, as well as $\mathsf{md}(\langle \tau, \tau' \rangle) = \mathbf{!}$ if $\mathsf{md}(\tau) = \mathbf{!}$ and $\mathsf{md}(\langle \tau, \tau' \rangle) = \updownarrow$ if $\mathsf{md}(\tau) = \downarrow$. We often write τ^p if $\mathsf{md}(\tau) = p$.

Quantifications bind type variables in pairs, so that both x and $\overline{\mathrm{x}}$ in τ are bound in $\forall \mathrm{x}.\tau$ and $\exists \mathrm{x}.\tau$. This extends to type substitution (which should always respect action modes), e.g. $(\overline{\mathrm{x}}(\mathrm{x})^{\uparrow})^{\mathbf{!}}[\tau/\mathrm{x}]$ is $(\overline{\tau}(\tau)^{\uparrow})^{\mathbf{!}}$. $\mathsf{ftv}(\tau)$ is the set of free type variables in τ, automatically including their duals. τ is *closed* if $\mathsf{ftv}(\tau) = \emptyset$.

3.3 Typing

We present a polymorphic type discipline based on implicit typing. The full version [6] explores different presentations and variants, including explicitly typed ones. The sequents have the form $\vdash_{\phi} P \triangleright A$, where A is an *action type*, a finite map from names to channel types, and ϕ is an *IO-mode*, which is either $\mathtt{I}$ or $\mathtt{O}$. In $\vdash_{\phi} P \triangleright A$, A assign types to free names in P, while ϕ indicates either P has an active thread ($\mathtt{O}$) or not ($\mathtt{I}$). We use the following operations and relations:

$$\frac{}{\vdash_{\mathtt{I}} \mathbf{0} \triangleright \emptyset}\ (\mathsf{Zero}) \qquad \frac{\vdash_{\phi_i} P_i \triangleright A_i \quad (i=1,2) \qquad A_1 \asymp A_2 \quad \phi_1 \asymp \phi_2}{\vdash_{\phi_1 \odot \phi_2} P_1 \mid P_2 \triangleright A_1 \odot A_2}\ (\mathsf{Par})$$

$$\frac{\vdash_{\phi} P \triangleright A, x:\tau \qquad \mathsf{md}(\tau) \in \{\mathbf{!}, \updownarrow\}}{\vdash_{\phi} (\boldsymbol{\nu}\, x)P \triangleright A}\ (\mathsf{Res}) \qquad \frac{\mathsf{md}(\tau) \in \{\mathbf{?}, \updownarrow\} \qquad \vdash_{\phi} P \triangleright A^{-x}}{\vdash_{\phi} P \triangleright A, x:\tau}\ (\mathsf{Weak})$$

$$\frac{\vdash_{\mathtt{0}} P \triangleright \vec{y}:\vec{\tau}, \uparrow ?A^{-x}}{\vdash_{\mathtt{I}} x(\vec{y}).P \triangleright x:\forall \vec{\mathrm{X}}.(\vec{\tau})^{\downarrow}, A}\ (\mathsf{In}^{\downarrow})\ (\mathrm{x}_i \notin \mathsf{ftv}(A)) \qquad \frac{\vdash_{\mathtt{0}} P \triangleright \vec{y}:\vec{\tau}, ?A^{-x}}{\vdash_{\mathtt{I}} !x(\vec{y}).P \triangleright x:\forall \vec{\mathrm{X}}.(\vec{\tau})^{!}, A}\ (\mathsf{In}^{!})\ (\mathrm{x}_i \notin \mathsf{ftv}(A))$$

$$\frac{\tau_i' = \tau_i[\rho_i/\mathrm{x}_i]}{\vdash_{\mathtt{0}} \overline{x}\langle\vec{y}\rangle \triangleright x:\exists \vec{\mathrm{X}}.(\vec{\tau})^{p_{\mathrm{O}}}, \vec{y}:\overline{\vec{\tau'}}}\ (\mathsf{Out})\ (\mathrm{x}_i \notin \mathsf{ftv}(\vec{\tau}'))$$

Fig. 1. Polymorphic Sequential Typing

- $\odot$ on IO-modes is a partial operation given by: $\mathtt{I} \odot \mathtt{I} = \mathtt{I}$ and $\mathtt{I} \odot \mathtt{0} = \mathtt{0} \odot \mathtt{I} = \mathtt{0}$ (note $\mathtt{0} \odot \mathtt{0}$ is not defined). When $\phi_1 \odot \phi_2$ is defined we write $\phi_1 \asymp \phi_2$.
- $\odot$ on channel types is the least commutative partial operation such that: (1) $\tau_{\mathtt{I}} \odot \overline{\tau_{\mathtt{I}}} = \langle \tau_{\mathtt{I}}, \overline{\tau_{\mathtt{I}}} \rangle$ and (2) $\tau \odot \tau = \tau$ and $\langle \overline{\tau}, \tau \rangle \odot \tau = \langle \overline{\tau}, \tau \rangle$ ($\mathsf{md}(\tau) = \mathbf{?}$). Then $A \asymp B$ iff $\tau' \odot \tau''$ is defined whenever $x:\tau' \in A$ and $x:\tau'' \in B$. If $A \asymp B$, then we set $A \odot B = (A \backslash B) \cup (B \backslash A) \cup \{x:\tau \mid x:\tau' \in A, x:\tau'' \in B, \tau = \tau' \odot \tau''\}$.

$\phi_1 \asymp \phi_2$ ensures that a well-typed process has at most one thread, while $A \asymp B$ guarantees determinism. A, B is the union of A and B, assuming their domains are disjoint; A^{-x} means A such that $x \notin \mathsf{fn}(A)$; and $\vec{p}A$ indicates $\mathsf{md}(A) \subset \{\vec{p}\}$.

The typing rules are given in Figure 1, which follow structure of processes except (Weak). In (Out), we assume $y_i = y_j$ implies $\tau_i = \tau_j$, $\rho_i = \rho_j$ and $\mathrm{x}_i = \mathrm{x}_j$. $\overline{\vec{\tau}}$ is the pointwise dualisation of $\vec{\tau}$. In comparison with the first-order affine typing [7], the only difference is introduction of quantifiers in ($\mathsf{In}^{\downarrow,!}$) and (Out), each with a natural variable condition. This prefix-wise quantification, close to the one adopted in [30], quantifies only input types (resp. output types) universally (resp. existentially). More general forms of polymorphic typing exist, which are studied in [6]: this form however has the merit in that it is syntactically tractable while harnessing enough expressive power for many practical purposes. Below we list simple examples of polymorphic processes (expressions are from Section 2), followed by a basic syntactic result. Henceforth $\twoheadrightarrow$ stands for $\equiv \cup \longrightarrow^*$.

Example 1. 1. Let $\mathbb{I} \stackrel{\text{def}}{=} x:\forall \mathrm{X}.(\overline{\mathrm{X}}^{\mathbf{?}}(\mathrm{X}^{\mathbf{!}})^{\uparrow})^{\mathbf{!}}$. Then $\vdash_{\mathtt{I}} \mathsf{id}\langle x\rangle \triangleright x:\mathbb{I}$.

2. $\vdash_{\mathtt{I}} \mathsf{t}\langle x\rangle \triangleright x : \mathbb{B}$, $\vdash_{\mathtt{I}} \mathsf{f}\langle x\rangle \triangleright x : \mathbb{B}$ and $\vdash_{\mathtt{I}} \mathsf{not}\langle xy\rangle \triangleright x : \mathbb{B}, y : \overline{\mathbb{B}}$. Further let $\mathsf{if}\ x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \stackrel{\text{def}}{=} \overline{x}(b_1 b_2 z)(!b_1(\vec{v}a).P_1 | !b_2(\vec{v}a).P_2 | z(\underline{b}).\overline{b}\langle \vec{v}a\rangle)$ assuming $\vdash_{\mathtt{0}} P_{1,2} \triangleright \vec{v}:\vec{\tau}^{\mathbf{?}}, a:\tau^{\uparrow}$. Then $\vdash_{\mathtt{0}} \mathsf{if}\ x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \triangleright x:\overline{\mathbb{B}}, \vec{v}:\vec{\tau}^{\mathbf{?}}, a:\tau^{\uparrow}$. We can check $\mathsf{if}\ x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \mid \mathsf{t}\langle x\rangle \twoheadrightarrow P_1 | \mathsf{t}\langle x\rangle | (\boldsymbol{\nu}\, b_2)!b_2(\vec{v}a).P_2 \approx P_1 | \mathsf{t}\langle x\rangle$ where $\approx$ is the standard (untyped) weak bisimilarity.

Proposition 1. (subject reduction) *$\vdash_{\phi} P \triangleright A$ and $P \twoheadrightarrow P'$ imply $\vdash_{\phi} P' \triangleright A$.*

3.4 Extension (1): Linearity

The first-order linear typing [32] refines the affine type discipline [7] by adding causality edges between typed names in an action type. Edges prevent circular causality. For example, $a.\overline{b} \,|\, b.\overline{a}$ is typable in the affine system, but not in the linear one. Its second-order extension simply adds prefix-wise quantification to [32]. Thus, in Figure 1, the input prefix rules become, with $x:\tau \to A$ denoting the result of adding a new edge from $x:\tau$ to each maximal node in A:

$$\frac{(\mathsf{In}^{\downarrow})\quad (\mathrm{X}_i \notin \mathsf{ftv}(A,B)) \qquad \vdash_{\mathtt{0}} P \triangleright \vec{y}:\vec{\tau}\,, \uparrow A^{-x}, ?B^{-x}}{\vdash_{\mathtt{I}} x(\vec{y}).P \triangleright (x:\forall \vec{\mathrm{X}}.(\vec{\tau})^{\downarrow} \to A), B} \qquad \frac{(\mathsf{In}^{!})\quad (\mathrm{X}_i \notin \mathsf{ftv}(A)) \qquad \vdash_{\mathtt{0}} P \triangleright \vec{y}:\vec{\tau}, ?A^{-x}}{\vdash_{\mathtt{I}} !x(\vec{y}).P \triangleright x:\forall \vec{\mathrm{X}}.(\vec{\tau})^{!} \to A}$$

Further $\asymp$ and $\odot$ in (Par) are refined to prohibit circularity of causal chains following [32]. The resulting system preserves all key properties of the first-order linear typing, but with greater typability. We state one of the central results.

Theorem 1. (strong normalisability) *Let $\vdash_{\phi} P \triangleright A$ in linear polymorphic typing. Then P is strongly normalising with respect to $\longrightarrow$.*

3.5 Extension (2): State and Concurrency

The integration of imperative features and polymorphism is an old and challenging technical problem [10, 16, 29]. Here we present a basic extension of affine polymorphic processes to stateful computation. Following [12], we add a constant process $\mathsf{Ref}\langle xy\rangle$, called *reference agent*. For interacting with reference, we need *selection* $\overline{x}\mathtt{in}_i\langle\vec{z}\rangle$ which selects, in the case of reference, either read ($i=1$) or write ($i=2$). For reduction we have:

$$\mathsf{Ref}\langle xy\rangle|\overline{x}\mathtt{in}_1\langle c\rangle \longrightarrow \mathsf{Ref}\langle xy\rangle|\overline{c}\langle y\rangle \qquad \mathsf{Ref}\langle xy\rangle|\overline{x}\mathtt{in}_2\langle zc\rangle \longrightarrow \mathsf{Ref}\langle xz\rangle|\overline{c}$$

The first rule describes reading of the content y, the second one writing of a new content z. A significant property of reference agents is that, in combination with replication, they can represent a large class of stateful computation [2, 12].

For types, we add the mutable replication mode $!_{\mathrm{M}}$ and its dual $?_{\mathrm{M}}$, as well as adding $[\&_i \vec{\tau}_{\mathtt{0}i}]^{p_{\mathtt{I}}}$ for input types and $[\oplus_i \vec{\tau}_{\mathtt{I}i}]^{p_{\mathtt{0}}}$ for output types. For example, the type of a reference with values of type τ is $[(\tau)^{\uparrow}\&\overline{\tau}()^{\uparrow}]^{!_{\mathrm{M}}}$, which we write $\mathsf{ref}(\tau)$. There are several ways to incorporate polymorphism into mutable types. Here we present a most basic form. Let us say $\forall \mathrm{X}.\tau$ (resp. $\exists \mathrm{X}.\tau$) is *simple* when $\mathsf{md}(\tau) \neq !_{\mathrm{M}}$ (resp. $\mathsf{md}(\tau) \neq ?_{\mathrm{M}}$). We then restrict the set of polymorphic types which we consider to the simple ones, and introduce the following typing rules.

$$\frac{(\mathsf{Ref})\quad \mathsf{md}(\tau) \in \{!, !_{\mathrm{M}}\}}{\vdash_{\mathtt{I}} \mathsf{Ref}\langle xy\rangle \triangleright x:\mathsf{ref}(\tau), y:\overline{\tau}} \qquad \frac{(\mathsf{Sel})\quad -}{\vdash_{\mathtt{0}} \overline{x}\mathtt{in}_i\langle\vec{y}\rangle \triangleright x:[\oplus_i \vec{\tau}_i]^{p_{\mathrm{O}}}, \vec{y}:\vec{\tau}_i} \qquad \frac{(\mathsf{In}^{!_{\mathrm{M}}})\quad \vdash_{\mathtt{0}} P \triangleright \vec{y}:\vec{\tau}, ?_{\mathrm{M}}?A^{-x}}{\vdash_{\mathtt{I}} !x(\vec{y}).P \triangleright x:(\vec{\tau})^{!_{\mathrm{M}}}, A}$$

Note $(\mathsf{In}^{!_{\mathrm{M}}})$ allows a replicated prefix to suppress $?_{\mathrm{M}}$-actions, unlike $(\mathrm{In}^{!})$. Also note the subject of a reference/$!_{\mathrm{M}}$-typed replication is never universally abstracted, in accordance with restriction to simple types. In spite of this limitation,

a wide variety of imperative polymorphic programs are typable via encoding: for example, all benchmark programs in Leroy's thesis [16] as well as Grossman's integrations of `struct` with existentials [10] are typable. This is due to the distinction between two replicated types, $!$ and $!_{\mathrm{M}}$. For further discussions, see [6]. For incorporating concurrency, we simply ignore all IO-modes in each rule. Section 6 presents equational reasoning for stateful polymorphic processes.

4 Contextual Congruence and Parametericity

This section presents a sound and complete characterisation of the contextual congruence by a second-order logical relation for the affine polymorphic π-calculus. As a consequence we obtain relational parametricity [28] and simulation principle [19, 27], the two fundamental principles for polymorphic λ-calculi.

The contextual congruence for affine polymorphic processes is defined following its first-order counterpart [7]. Write $\mathbb{O}$ for $()^{\uparrow}$ and write $P \Downarrow_x$ when $P \longrightarrow^* (\boldsymbol{\nu}\, \vec{z})(\overline{x}\langle\vec{y}\rangle|P')$ with $x \notin \{\vec{z}\}$. Then $\cong_{\forall\exists}$ is the maximum typed congruence over polymorphic processes satisfying

$$\vdash_{\mathtt{0}} P_1 \cong_{\forall\exists} P_2 \triangleright x{:}\mathbb{O} \Leftrightarrow (P_1 \Downarrow_x \Leftrightarrow P_2 \Downarrow_x).$$

for all $\vdash_{\mathtt{0}} P_{1,2} \triangleright x{:}\mathbb{O}$. We write $P \cong^{A,\phi}_{\forall\exists} Q$ if P and Q are related by $\cong_{\forall\exists}$ under A, ϕ (and often omit ϕ or A, ϕ). We can easily check that $\equiv \cup \longrightarrow \subsetneq \cong_{\forall\exists}$.

We first consider logical relations in a simple shape. Given closed types $\tau_{1,2}$ with $\mathsf{md}(\tau_1) = \mathsf{md}(\tau_2) \in \{\uparrow, !\}$, a *typed relation* $\mathfrak{R} : \tau_1 \leftrightarrow \tau_2$ is a family of binary relations $\{\mathfrak{R}_x\}_{x\in\mathcal{N}}$ over typed processes such that: (1) if $P_1 \mathfrak{R}_x P_2$ then $\vdash_\phi P_i \triangleright x{:}\tau_i$ with $\phi = \mathtt{I}$ (resp. $\phi = \mathtt{0}$) if $\mathsf{md}(\tau_i) = !$ (resp. $\mathsf{md}(\tau_i) = \uparrow$) and (2) the family is closed under injective renaming, i.e. $P \mathfrak{R}_x Q$ iff $P\binom{xy}{yx} \mathfrak{R}_y\, Q\binom{xy}{yx}$.

Given a typed relation $\mathfrak{R} : \tau_1 \leftrightarrow \tau_2$, the *dual of* $\mathfrak{R}$ *at* xu, written $\mathfrak{R}^{\perp}_{xu}$, is a relation from processes of type $x{:}\overline{\tau_1}, u{:}\mathbb{O}$ to those of type $x{:}\overline{\tau_2}, u{:}\mathbb{O}$, satisfying: $P_1 \mathfrak{R}^{\perp}_{xu} P_2$ iff $(\boldsymbol{\nu}\, x)(P_1|R_1)\Downarrow_u \Leftrightarrow (\boldsymbol{\nu}\, x)(P_2|R_2)\Downarrow_u$ for each $R_1 \mathfrak{R}_x R_2$. The resulting relations, called *typed co-relations*, are also taken modulo injective renaming, so that we simply write $\mathfrak{R}^{\perp}$ for the dual of $\mathfrak{R}$. Symmetrically we define the dual of a co-relation, returning to a typed relation. A $\perp\!\!\perp$*-closed relation* is a typed relation closed under double negation, i.e. $\mathfrak{R}$ such that $\mathfrak{R}^{\perp\perp} = \mathfrak{R}$.

We can now define logical relations as interpretation of open types under a *relational environment*, i.e. a function which maps type variables to $\perp\!\!\perp$-closed relations respecting action modes. The interpretation is written $(\!(\tau)\!)_\xi$ where ξ is a relational environment.

$$(\!((\tau_1^? .. \tau_n^? \rho^{\uparrow})^!)\!)_\xi \stackrel{\text{def}}{=} ((\!(\overline{\tau_1})\!)_\xi .. (\!(\overline{\tau_n})\!)_\xi (\!(\rho)\!)_\xi)^! \qquad (\!((\tau_1 .. \tau_n)^{\uparrow})\!)_\xi \stackrel{\text{def}}{=} ((\!(\tau_1)\!)_\xi .. (\!(\tau_n)\!)_\xi)^{\uparrow}$$
$$(\!(\mathrm{X})\!)_\xi \stackrel{\text{def}}{=} \xi(\mathrm{X}) \quad (\!(\forall \mathrm{X}.\tau)\!)_\xi \stackrel{\text{def}}{=} \forall \mathrm{X}.\lambda\mathfrak{R}.(\!(\tau)\!)_{\xi\cdot \mathrm{X}\mapsto\mathfrak{R}^{\perp\perp}} \quad (\!(\exists \mathrm{X}.\tau)\!)_\xi \stackrel{\text{def}}{=} \exists \mathrm{X}.\lambda\mathfrak{R}.(\!(\tau)\!)_{\xi\cdot \mathrm{X}\mapsto\mathfrak{R}^{\perp\perp}}$$

Above, the right-hand side of each definition uses a type-respecting function on typed relations, given in the following (definitions are presented for simpler shapes for legibility, with obvious generalisations).

$$(\mathfrak{R}_1^{?}\mathfrak{R}_2^{\uparrow})^{!}_{x} \stackrel{\text{def}}{=} \{\langle P, P'\rangle \mid Q\ \mathfrak{R}_{1y}\ Q' \supset P \circ \overline{x}\langle yz\rangle \circ Q\ \mathfrak{R}_{2z}\ P' \circ \overline{x}\langle yz\rangle \circ Q'\}$$
$$(\mathfrak{R}^{!})^{\uparrow}_{x} \stackrel{\text{def}}{=} \{\langle \overline{x}\langle y\rangle \circ Q, \overline{x}\langle y\rangle \circ Q'\rangle \mid Q\mathfrak{R}_y Q'\}^{\perp\perp}$$
$$\forall \mathrm{X}.\mathfrak{R}[\mathrm{X}]_x \stackrel{\text{def}}{=} \{\langle P, P'\rangle \mid \forall \mathfrak{R}'.P\ \mathfrak{R}[\mathfrak{R}']_x\ P'\}$$
$$\exists \mathrm{X}.\mathfrak{R}[\mathrm{X}]_x \stackrel{\text{def}}{=} \{\langle P, P'\rangle \mid \exists \mathfrak{R}'.\ P\ \mathfrak{R}[\mathfrak{R}']_x\ P'\ \}^{\perp\perp},$$

where all mentioned processes, substitutions etc. should be appropriately typed. $P \circ Q$ denotes $(\boldsymbol{\nu}\ \mathsf{fn}(P) \cap \mathsf{fn}(Q))(P|Q)$. $\mathfrak{R}[\mathrm{X}]$ indicates a type-respecting map over typed relations.[3] These rules can be read quite like logical relations for functions: for example, the first rule says that, if a pair of "resources" are related, then the corresponding pair of "results" should also be related. In fact, the construction yields, via encoding, logical relations in the usual sense for both call-by-name and call-by-value polymorphic PCF-like calculi, cf. [25, 26]. Since each rule returns a $\perp\perp$-closed relation whenever its arguments are, $(\!(\tau)\!)_\xi$ is always $\perp\perp$-closed.

The above logical relation only relates processes with a single free name. For equating processes with multiple free names, we extend logical relations to action types which are *connected* in the following sense.

Definition 1. (A, ϕ) *is* connected *if one the following holds.*

- $\phi = \mathtt{I}$ *and* A *contains, in its range, either a unique* $!$*-type and zero or more* $?$*-types, or a unique* $\downarrow$*-type, a unique* $\uparrow$*-type and zero or more* $?$*-types.*
- $\phi = \mathtt{O}$ *and* A *contains a unique* $\uparrow$*-type and zero or more* $?$*-types.*

If (A, ϕ) *is connected, the name with the unique* $\uparrow/!$ *type is its* principal port*.*

Connectedness has both practical and theoretical significance. First, in many practical examples including the embedding of programming languages, it is often enough to consider processes of connected types. Second, any process of an arbitrary action type can always be decomposed canonically into connected processes, so that results about connected processes often easily extend to non-connected processes. We now generalise the logical relation to connected types.

Definition 2. *Let* (A, ϕ) *be connected with principal port* $x : \tau$ *and let* $\mathsf{fn}(A) \setminus \{x\} = \{y_j\}_{j\in J}$*. Then* $\cong_{\mathcal{L}}^{A,\phi}$ *is a relation on processes of type* (A, ϕ) *which relates* P *and* P' *iff, for each* ξ *(*$\prod_{j\in J} P_j$ *denotes a parallel composition of* $\{P_j\}_{j\in J}$*),*

$$(\forall j \in J.\ Q_j\ (\!(\overline{A(y_j)})\!)_{\xi,y_j} Q'_j)\ \supset\ (\boldsymbol{\nu}\ \vec{y})(P \mid \Pi_{j\in J} Q_j)\ (\!(\tau)\!)_{\xi,x}\ (\boldsymbol{\nu}\ \vec{y})(P' \mid \Pi_{j\in J} Q'_j).$$

Note that $\cong_{\mathcal{L}}^{x:\tau,\phi}$ (with ϕ given corresponding to τ) coincides with $(\!(\tau)\!)_x$. The following result is proved closely following the development by Pitts [25, 26].

Theorem 2. (characterisation of $\cong_{\forall\exists}$) $\cong_{\mathcal{L}}^{A,\phi} = \cong_{\forall\exists}^{A,\phi}$ *for each connected* (A, ϕ)*.*

Corollary 1. 1. (parametricity) $P \cong_{\forall\exists}^{x:\forall \mathrm{X}.\tau} Q$ *if and only if* $P(\!(\tau)\!)_{\mathrm{X}\mapsto\mathfrak{R}} Q$ *for each* $\perp\perp$*-closed* $\mathfrak{R}$*.*
2. (simulation) $P \cong_{\forall\exists}^{x:\exists \mathrm{X}.\tau} Q$ *if and only if* $P(\!(\tau)\!)_{\mathrm{X}\mapsto\mathfrak{R}} Q$ *for some* $\perp\perp$*-closed* $\mathfrak{R}$*.*

[3] In detail: $\mathfrak{R}[\mathrm{X}]$ should map, for fixed τ and τ' such that $\mathsf{ftv}(\tau) \cup \mathsf{ftv}(\tau') \subset \{\mathrm{X}\}$, each $\mathfrak{R}' : \rho \leftrightarrow \rho'$ of mode $\mathsf{md}(\mathrm{X})$ to a typed relation $\mathfrak{R}[\mathfrak{R}'] : \tau[\rho/\mathrm{X}] \leftrightarrow \tau'[\rho'/\mathrm{X}]$.

The construction and results extend to the whole set of affine polymorphic processes, see [6]. The same characterisation result also holds for linear polymorphic processes, where we use a $\bot\bot$-closure based on convergence to a specific boolean value (this convergence is also used for defining the contextual congruence, which is necessary since linear processes are always converging). In Section 6 we give reasoning examples which use these results. The corresponding characterisation results for non-functional polymorphic behaviour (including state [24] and control) are left as an open issue.

5 Generic Transitions and Innocence

This section discusses another basic element of the present theory, a generic labelled transition system and the induced process equivalence. While our presentation focusses on the affine polymorphic π-calculus, the construction equally applies to linear, stateful and concurrent polymorphic processes, with the same soundness result. The duality principle strongly guides the construction. The set of action labels $(l, l', \ldots)$ are given by:

$$l \ ::= \ x\langle(\vec{y})\vec{w}\rangle \mid \overline{x}\langle(\vec{y})\vec{w}\rangle \mid \boldsymbol{\tau}$$

In the first two labels, names in $\vec{y}$ are pairwise distinct and $\vec{y}$ is a (not necessarily consecutive) subsequence of $\vec{w}$ (called *objects*) and distinct from x (called *subject*). Names in $\vec{y}$ occur *bound*, while all other names occur *free*. $x(\vec{y})$ and $x\langle\vec{y}\rangle$ stand for $x\langle(\vec{y})\vec{y}\rangle$ and $x\langle(\varepsilon)\vec{y}\rangle$, respectively and similarly for output actions.

Transitions use an extended typing where type variables in action types are annotated by quantification symbols (as $\mathrm{X}^{\forall}$ and $\mathrm{X}^{\exists}$, called *universal type variable* and *existential type variable*, respectively). The original free type variables and $\forall$-quantified variables are naturally $\forall$-annotated, while $\exists$-quantified variables are $\exists$-annotated. Free $\exists$-type variables are introduced by the following added rule:

$$(\exists\text{-Var}) \ \frac{\vdash_{\phi} P \triangleright A[\tau/\mathrm{X}^{\exists}]}{\vdash_{\phi} P \triangleright A}$$

which we assume to be applicable only as the last rule(s) in a derivation. As an example of typing, we have $\vdash_{\mathsf{0}} \mathsf{t}\langle y\rangle|z(w).\overline{e}(c)\mathsf{not}\langle cw\rangle \triangleright y : \mathrm{X}^{\exists}, z : (\overline{\mathrm{X}}^{\exists})^{\downarrow}, e : (\mathbb{B})^{\uparrow}$, abstracting away the type which is both for the resource at y and for the value of the input via z. Using annotated type variables, the following predicates decide if the shape of action labels conforms to a given action type. In brief, they say that free output (resp. input) corresponds to universal type variables (resp. existential type variables), cf. Section 2. θ below denotes a sequence of quantifiers.

Definition 3. *1.* $A \vdash \boldsymbol{\tau}$ *always.*

2. $A \vdash x\langle(\vec{z})\vec{w}\rangle : \theta(\vec{\tau})^{p_{\mathrm{I}}}$ *when* $\{\vec{z}\} \cap \mathsf{fn}(A) = \emptyset$ *and* $A(x) = \theta(\vec{\tau})^{p_{\mathrm{I}}}$ *s.t.* $w_i \notin \{\vec{z}\}$ *iff* $A(w_i) = \overline{\tau_i}$ *where* τ_i *is an existential type variable.*

3. $A \vdash \overline{x}\langle(\vec{z})\vec{w}\rangle : \theta(\vec{\tau})^{p_{\mathrm{O}}}$ *when* $\{\vec{z}\} \cap \mathsf{fn}(A) = \emptyset$ *and* $A(x) = \theta(\vec{\tau})^{p_{\mathrm{O}}}$ *s.t.* $w_i \notin \{\vec{z}\}$ *iff* $A(w_i) = \overline{\tau_i}$ *where* τ_i *is a universal type variable.*

We can now introduce the transition rules (for expository purposes we focus on key instances). We start from the standard bound input.

$$(\mathsf{BIn}^{\downarrow}) \quad \vdash_{\mathtt{I}} x(\vec{y}).P \triangleright A \xrightarrow{x(\vec{y})} \vdash_{\mathtt{O}} P \triangleright A/x, \vec{y}{:}\vec{\tau} \qquad (A \vdash x(\vec{y}) : \theta(\vec{\tau})^{\downarrow})$$

This may introduce (output-moded) $\forall$-type variables, which are used as follows.

$$(\mathsf{FOut}^{\uparrow}) \quad \vdash_{\mathtt{O}} \overline{x}\langle\vec{y}\rangle \triangleright A \xrightarrow{\overline{x}\langle\vec{y}\rangle} \vdash_{\mathtt{I}} \mathbf{0} \triangleright A/x \qquad (A \vdash \overline{x}\langle\vec{y}\rangle : \theta(\vec{\mathrm{X}}^{\forall})^{\uparrow})$$

We can now infer $\vdash_{\mathtt{I}} \mathsf{id}\langle x\rangle \triangleright x{:}\mathbb{I} \xrightarrow{x(yz)} \xrightarrow{\overline{z}\langle y\rangle} \vdash_{\mathtt{I}} \mathsf{id}\langle x\rangle \triangleright x{:}\mathbb{I}$ (using a replicated version for input). Next we consider the dual situation, starting from bound output.

$$(\mathsf{BOut}^{\uparrow}) \ \vdash_{\mathtt{O}} \overline{x}\langle\vec{y}\rangle \triangleright A \xrightarrow{\overline{x}(\vec{z})} \vdash_{\mathtt{I}} \Pi_i[z_i \to y_i]^{\tau_i} \triangleright A/x, \vec{z} : \vec{\tau} \quad (A \vdash \overline{x}(\vec{z}) : \theta(\vec{\overline{\tau}})^{\uparrow})$$

Here $[z_i \to y_i]^{\tau_i}$ is the standard copy-cat agent [7, 12, 14, 32, 33]. For example, $[a \to b]^{(()^{\uparrow})^!} \stackrel{\text{def}}{=} !a(y).\overline{b}(y')y'.\overline{y}$. This rule is best seen in view of the semantic equality $\overline{x}\langle\vec{y}\rangle \cong \overline{x}(\vec{z})\Pi_i[z_i \to y_i]^{\tau_i}$. Again this rule may introduce (input-moded) $\exists$-type variables, used by:

$$(\mathsf{FIn}^{\downarrow}) \ \vdash_{\mathtt{I}} x(\vec{y}).P \triangleright A \xrightarrow{x\langle\vec{z}\rangle} \vdash_{\mathtt{O}} P\{\vec{z}/\vec{y}\} \triangleright A/x \odot \vec{z}{:}\mathrm{X} \quad (A \odot \vec{z}{:}\overline{\vec{\mathrm{X}}}^{\exists} \vdash x\langle\vec{z}\rangle : \theta(\vec{\mathrm{X}}^{\exists})^{\downarrow})$$

In the side condition, we compose types for opaque resources to appear in a later derivation. The rule says an input may receive channels for opaque resources which have been exported and which are, therefore, free. We can now infer $\vdash_{\mathtt{O}} \overline{x}(yz)(\mathsf{t}\langle y\rangle | z(w).R) \triangleright x{:}\overline{\mathbb{I}}, e{:}(\mathbb{B})^{\uparrow} \xrightarrow{\overline{x}(yz)} \xrightarrow{z\langle y\rangle} \vdash_{\mathtt{O}} \mathsf{t}\langle y\rangle | R\{y/w\} \triangleright y{:}\langle \mathrm{X}^{\exists}, \overline{\mathrm{X}}^{\exists}\rangle, e{:}(\mathbb{B})^{\uparrow}$.

Since a type may carry both type variable(s) and concrete type(s), the general rule for linear input (resp. output) combines $(\mathsf{BIn}^{\downarrow})$ and $(\mathsf{FIn}^{\downarrow})$ (resp. $(\mathsf{BOut}^{\uparrow})$ and $(\mathsf{FOut}^{\uparrow})$). Similarly we have rules for replicated input/output, as well as standard composition rules. For the generated transition relation we can check, under the extended typing:

Proposition 2. *If* $\vdash_{\phi} P \triangleright A$ *and* $\vdash_{\phi} P \triangleright A \xrightarrow{l} \vdash_{\phi'} Q \triangleright B$ *then* $\vdash_{\phi'} Q \triangleright B$.

Define the weak bisimilarity $\approx_{\forall\exists}$ induced by generic transitions in the standard way. The proof of the following is then straightforward.

Proposition 3. (soundness) $\vdash_{\phi} P \approx_{\forall\exists} Q \triangleright A$ *implies* $\vdash_{\phi} P \cong_{\forall\exists} Q \triangleright A$.

The result extends to the linear/stateful extensions in §3.4/5. Further the analogue of Corollary 1(1) (parametricity) easily holds for $\approx_{\forall\exists}$. We can also show polymorphic transition sequences of a typed process can be characterised by an innocent function as in the first-order affine processes [7]. Again as in [7], finite generic innocent functions are always realisable as syntactic processes.

6 Reasoning Examples

This section discusses equational reasoning based on the theories in Sections 4 and 5, and outlines a fully abstract embedding of System F.

Inhabitation Results. We begin with an inhabitation result for $\mathbb{I}$ using generic transitions. Let $\Omega\langle x\rangle \stackrel{\text{def}}{=} !x(yz).(\boldsymbol{\nu}\, ab)(!a(w).\overline{b}\langle w\rangle|!b(w).\overline{a}\langle w\rangle|\overline{a}(c)c.\overline{z}\langle y\rangle)$ (which diverges after the initial input; from now on, the notation $\Omega\langle x\rangle$ is used for denoting such processes regardless of types). We prove that $\vdash_{\mathtt{I}} P \triangleright x:\mathbb{I}$ implies either $P \approx_{\forall\exists} \mathsf{id}\langle x\rangle$ or $P \approx_{\forall\exists} \Omega\langle x\rangle$. Let $\vdash_{\mathtt{I}} P \triangleright x:\mathbb{I}$. Then we have $\vdash_{\mathtt{I}} P \triangleright x:\mathbb{I} \stackrel{x(yz)}{\longrightarrow} \vdash_{\mathtt{O}} P' \triangleright x:\mathbb{I}, y:\mathrm{X}^{\forall}, z:(\mathrm{X}^{\forall})^{\uparrow}$. By inspecting the action type, if P' ever has an output, it can only be $\overline{z}\langle y\rangle$, in which case $P \approx_{\forall\exists} \mathsf{id}\langle x\rangle$. If not then $P \approx_{\forall\exists} \Omega\langle x\rangle$. Since $\mathsf{id}\langle x\rangle \not\approx_{\forall\exists} \Omega\langle x\rangle$, these two are all distinct inhabitants of the type. Similarly we can check $x:\mathbb{B}$ is inhabited by $\mathtt{t}\langle x\rangle$, $\mathtt{f}\langle x\rangle$ and $\Omega\langle x\rangle$. In the linear typing, we obtain the same results except we lose $\Omega\langle x\rangle$ by totality of transition.

Boolean ADTs. Next we show a simple use of logical relations for equational reasoning, taking abstract data types of opaque booleans (similar to those discussed in [22, 25]). The data type should export a "flip", or negation operation and allow reading (which means turning an opaque boolean to a concrete one). Two simple implementations in the λ-calculus with records are:

$$M \stackrel{\text{def}}{=} \mathtt{pack}\ \mathtt{bool}\ \{\mathtt{bit} = \mathsf{T},\ \mathtt{flip} = \lambda x{:}\mathtt{bool}.\neg x,\ \mathtt{read} = \lambda x{:}\mathtt{bool}.x\ \}\ \mathtt{as}\ \mathit{bool}$$
$$M' \stackrel{\text{def}}{=} \mathtt{pack}\ \mathtt{bool}\ \{\mathtt{bit} = \mathsf{F},\ \mathtt{flip} = \lambda x{:}\mathtt{bool}.\neg x,\ \mathtt{read} = \lambda x{:}\mathtt{bool}.\neg x\}\ \mathtt{as}\ \mathit{bool}$$

where $\mathit{bool} \stackrel{\text{def}}{=} \exists \mathrm{X}.\{\mathtt{bit} : \mathrm{X},\ \mathtt{flip} : \mathrm{X} \to \mathrm{X}, \mathtt{read} : \mathrm{X} \to\ \mathtt{bool}\}$. M and M' can be encoded as (using a call-by-value translation of products, cf. [32]):

$$\mathtt{bool}\langle u\rangle \stackrel{\text{def}}{=} \overline{u}(m_1m_2m_3)(Q_1|Q_2|Q_3) \qquad \mathtt{bool}'\langle u\rangle \stackrel{\text{def}}{=} \overline{u}(m_1m_2m_3)(Q_1'|Q_2'|Q_3')$$

where $Q_1 \stackrel{\text{def}}{=} \mathtt{t}\langle m_1\rangle$, $Q_2 \stackrel{\text{def}}{=} !m_2(bz).\overline{z}(b')\mathsf{not}\langle b'b\rangle$, $Q_3 \stackrel{\text{def}}{=} !m_3(bz).\overline{z}\langle b\rangle$, $Q_1' \stackrel{\text{def}}{=} \mathtt{f}\langle m_1\rangle$, $Q_2' \equiv Q_2$ and $Q_3' \stackrel{\text{def}}{=} !m_3(bz).\overline{z}(b')\mathsf{not}\langle b'b\rangle$. We can easily check these processes are typable under $u:\exists \mathrm{X}.\mathcal{B}[\mathrm{X}]$, where $\mathcal{B}[\mathrm{X}] \stackrel{\text{def}}{=} (\mathrm{X}(\overline{\mathrm{X}}(\mathrm{X})^{\uparrow})^{\mathbf{!}}(\overline{\mathrm{X}}(\mathbb{B})^{\uparrow})^{\mathbf{!}})^{\uparrow}$.

We now show $\vdash_{\mathtt{I}} \mathtt{bool}\langle u\rangle \cong_{\forall\exists} \mathtt{bool}'\langle u\rangle \triangleright x:\exists \mathrm{X}.\mathcal{B}[\mathrm{X}]$. By Corollary 1(2), it is enough to establish $\mathtt{bool}\langle u\rangle(\!(\mathcal{B}[\mathrm{X}])\!)_{x,\mathrm{X}\mapsto\mathfrak{R}}\mathtt{bool}'\langle u\rangle$ for some $\bot\!\bot$-closed $\mathfrak{R}$. By definition this means we have to verify:

$$Q_1\mathfrak{R}_{m_1}Q_1', \quad Q_2(\overline{\mathfrak{R}}(\mathfrak{R})^{\uparrow})^{\mathbf{!}}_{m_2}Q_2', \quad Q_3(\overline{\mathfrak{R}}(\!(\mathbb{B})\!)^{\uparrow})^{\mathbf{!}}_{m_3}Q_3'.$$

Take $\mathfrak{R} \stackrel{\text{def}}{=} \{(\mathtt{t}\langle x\rangle, \mathtt{f}\langle x\rangle),\ (\mathtt{f}\langle x\rangle, \mathtt{t}\langle x\rangle),\ (\Omega\langle x\rangle, \Omega\langle x\rangle)\}$ (processes are taken up to $\cong_{\forall\exists}$). Then $\mathfrak{R}$ is $\bot\!\bot$-closed (by the inhabitation result for $\mathbb{B}$). $\mathfrak{R}$ obviously relates Q_1 and Q_1'. The key case is $Q_3(\overline{\mathfrak{R}}(\!(\mathbb{B})\!)^{\uparrow})^{\mathbf{!}}_{m_3}Q_3'$, which means, by definition, $Q_3 \circ \overline{m}_3\langle xw\rangle \circ S\,(\!(\mathbb{B})\!)^{\uparrow}_{w}\, Q_3' \circ \overline{m}_3\langle xw\rangle \circ S'$ for any $S\mathfrak{R}S'$. The case when $(S, S') = (\Omega\langle x\rangle, \Omega\langle x\rangle)$ is trivial. Let $(S, S') = (\mathtt{t}\langle x\rangle, \mathtt{f}\langle x\rangle)$. We can check both $Q_3 \circ \overline{m}_3\langle xw\rangle \circ S$ and $Q_3' \circ \overline{m}_3\langle xw\rangle \circ S'$ reduce to, hence are $\cong_{\forall\exists}$-equivalent to, $\overline{w}\langle b\rangle \circ \mathtt{t}\langle b\rangle$. Now we use Theorem 2. Similarly when $(S, S') = (\mathtt{f}\langle x\rangle, \mathtt{t}\langle x\rangle)$. Reasoning for Q_2 and Q_2' is similar.

Simple Boolean Agent. In Section 2, we have seen the behaviour of $S \stackrel{\text{def}}{=} \overline{x}(yz)(\mathtt{t}\langle y\rangle|z(w).\overline{e}(b)\mathsf{not}\langle bw\rangle)$ under $x:\overline{\mathbb{I}}, e:(\mathbb{B})^{\uparrow}$. Noting this process is typable

in the linear typing, we show that S and $S' \stackrel{\text{def}}{=} \overline{e}(b)\mathsf{f}\langle b\rangle$ are contextually congruent as linear polymorphic processes. Since S and S' have different visible traces, the use of some extensionality principle is essential. By the characterisation result along the line of Theorem 2 in the linear setting, it suffices to show $(\boldsymbol{\nu}\, x)(S|P)(((\mathbb{B})^{\uparrow}))_e(\boldsymbol{\nu}\, x)(S'|P)$ for each $\vdash_{\mathtt{I}} P \triangleright x{:}\mathbb{I}$. But if $\vdash_{\mathtt{I}} P \triangleright x{:}\mathbb{I}$ then $P \cong_{\forall\exists} \mathsf{id}\langle x\rangle$ by inhabitation. We can then check $(\boldsymbol{\nu}\, x)(S|P) \cong_{\forall\exists} (\boldsymbol{\nu}\, x)(S|\mathsf{id}\langle x\rangle) \approx S' \approx (\boldsymbol{\nu}\, x)(S'|\mathsf{id}\langle x\rangle) \cong_{\forall\exists} (\boldsymbol{\nu}\, x)(S'|P)$, hence done.

Diverging Functions. Another example which needs extensionality, but this time in the context of affine sequential processes, is the equality of two diverging functions treated by Pitts [25] (the example is attributed to Stark). Assume we are given the following two call-by-value functions:

$$F' \stackrel{\text{def}}{=} \mathtt{letrec}\ f = \lambda g.fg\ \mathtt{in}\ f$$
$$G' \stackrel{\text{def}}{=} \mathtt{letrec}\ f = \lambda g.\mathtt{if}\ g\mathsf{T}\ \mathtt{then}\ (\mathtt{if}\ g\mathsf{F}\ \mathtt{then}\ fg\ \mathtt{else}\ \mathsf{T})\ \mathtt{else}\ fg\ \mathtt{in}\ f$$

Let $\mathsf{null}_\lambda \stackrel{\text{def}}{=} \forall \mathrm{X}.\mathrm{X}$, $\mathbb{B}_\lambda \stackrel{\text{def}}{=} \forall \mathrm{X}.(\mathrm{X}\Rightarrow\mathrm{X}\Rightarrow\mathrm{X})$ and $\alpha = \exists \mathrm{X}.((\mathrm{X}\Rightarrow\mathbb{B}_\lambda)\Rightarrow\mathbb{B}_\lambda)$. Then we can check $F \stackrel{\text{def}}{=} \mathtt{pack}\ \mathsf{null}_\lambda, F'\ \mathtt{as}\ \alpha$ and $G \stackrel{\text{def}}{=} \mathtt{pack}\ \mathbb{B}_\lambda, G'\ \mathtt{as}\ \alpha$ are well-typed after existential abstraction. To show F and G are equal, we first encode them as affine polymorphic processes. In the standard encoding (with recursion being translated using copy-cats), F and G are represented by, respectively, $\overline{u}(x)\Omega\langle x\rangle$ and $\overline{u}(x)P$ where (using some shorthand notations):

$$P \stackrel{\text{def}}{=} !x(gz).(\overline{g}(\mathsf{T}w)w(b).\mathsf{if}\ b\ \mathsf{then}\ [\overline{g}(\mathsf{F}w')w'(b').\mathsf{if}\ w'\ \mathsf{then}\ \mathsf{else}\ \Omega\langle u\rangle\overline{z}(\mathsf{T})]\ \mathsf{else}\Omega\langle u\rangle),$$

both typable under $u : \exists \mathrm{X}.(\tau)^{\uparrow}$ with $\tau = ((\mathrm{X}^{!}(\overline{\mathbb{B}})^{\downarrow})^{?}(\mathbb{B})^{\uparrow})^{!}$. We can then show $P \cong_{\forall\exists} \Omega\langle x\rangle$ using a logical relation $((\mathfrak{R}(\overline{\mathbb{B}})^{\downarrow})^{?}(\mathbb{B})^{\uparrow})^{!}_{x}$ where $\mathfrak{R}_u$ is the universal relation over $u{:}\mathbb{B}$. Detailed reasoning is given in [6].

State and Concurrency. We apply transition-based reasoning to a simple concurrent ADT, a cell with a boolean value. It allows three operations, `share`, `read` and `write`. The first returns the access pointer to the cell, while the latter two read/write a boolean value from it. The data type of this agent is:

$$\mathtt{Cell}[\mathbb{B}] \stackrel{\text{def}}{=} \exists \mathrm{X}.(((\mathrm{X})^{\uparrow})^{!_{\mathrm{M}}}(\overline{\mathrm{X}}(\mathbb{B})^{\uparrow})^{!_{\mathrm{M}}}(\overline{\mathrm{X}}\,\overline{\mathbb{B}}()^{\uparrow})^{!_{\mathrm{M}}})^{\uparrow}.$$

Below we give two implementations. The first is centralised in that all clients have access to a single container; while, in the second, each client has a different proxy which it uses to access the "real" cell. Let

$$\mathsf{cell}\langle ul\rangle \stackrel{\text{def}}{=} \overline{u}(srw)(S \mid R \mid W) \qquad \mathsf{cell}'\langle ul\rangle \stackrel{\text{def}}{=} \overline{u}(srw)(S' \mid R' \mid W')\ ,$$

where $S \stackrel{\text{def}}{=} !s(z).\overline{z}\langle l\rangle$, $R \stackrel{\text{def}}{=} !r(cz).\overline{c}\mathtt{in}_1(e)e(x).\overline{z}\langle x\rangle$ and $W \stackrel{\text{def}}{=} !w(cbz).\overline{c}\mathtt{in}_2\langle bz\rangle$, while $S' \stackrel{\text{def}}{=} !s(z).\overline{z}(c)!c(z').\overline{z'}\langle l\rangle$, $R' \stackrel{\text{def}}{=} !r(cz).\overline{c}(e)e(r').\overline{r'}\mathtt{in}_1(f)f(x).\overline{z}\langle x\rangle$ and

$W' \stackrel{\text{def}}{=} !w(cbz).\overline{c}(w)w(r).\overline{r}\mathtt{in}_2\langle bz\rangle$. Then both $\mathsf{cell}\langle ul\rangle$ and $\mathsf{cell}'\langle ul\rangle$ are typable under $u:\mathtt{Cell}[\mathbb{B}], l:\overline{\mathsf{ref}(\mathbb{B})}$, with $\mathsf{ref}(\tau) \stackrel{\text{def}}{=} [(\tau)^\uparrow \& \overline{\tau}()^\uparrow]^{!_{\mathrm{M}}}$.

To show these two typed processes are $\approx_{\forall\exists}$-equivalent, we first note that neither manipulates boolean values non-trivially (they are *data independent* in the sense of [15]), hence both are also typable under $u:\mathtt{Cell}[\mathrm{Y}^\forall], l:\overline{\mathsf{ref}(\mathrm{Y}^\forall)}$. By parametricity of $\approx_{\forall\exists}$, it suffices to consider a bisimulation under this typing, which radically reduces the number of transitions. We now construct a relation $\mathcal{R}$ from the following tuples:

$$\begin{array}{l}\vdash\ (S \mid R \mid W \mid \Pi_i[c_i \to l]^{\mathsf{ref}(\mathrm{Y})})\ \ \mathcal{R}\ \ (S' \mid R' \mid W' \mid \Pi_i !c_i(z).\overline{z}\langle l\rangle) \\ \qquad\qquad \triangleright\ s:((\mathrm{X}^\exists)^\uparrow)^{!_{\mathrm{M}}}, r:(\overline{\mathrm{X}}^\exists(\mathrm{Y}^\forall)^\uparrow)^{!_{\mathrm{M}}}, w:(\overline{\mathrm{X}}^\exists\overline{\mathrm{Y}}^\forall()^\uparrow)^{!_{\mathrm{M}}},\ \vec{c}:\vec{\mathrm{X}}^\exists\end{array}$$

together with their derivatives ([6] gives details). Note that $\Pi_i[c_i \to l]$ on the left-hand side is generated since S is in fact *not* allowed to do a free output via z (because l is not typed by a universal type variable; though it *is* typed by an existential variable). Observing that $c_i:\mathrm{X}_i^\exists$ prohibits each c_i from being used as the subject of an action, while permitting its use as an object of a free input (via r and w) that in turn triggers appropriate internal reduction, we can verify $\mathcal{R}$ is a bisimulation.

Fully Abstract Embedding of System F. Using the characterisation of polymorphic transitions by innocence mentioned in Section 5, we can embed System F (the second-order λ-calculus) fully abstractly in linear polymorphic processes. The contextual equality over λ-terms is defined in the standard way [20], using observables at the polymorphic boolean type. We write $M, N, \ldots$ for polymorphic λ-terms, $\alpha, \beta, \ldots$ for their types, and $\cong_\forall$ for the contextual equality. We can use different encodings to reach the same result: for example we can use Turner's call-by-value encoding [30] (other encodings, including those based on call-by-name, are discussed in [6]). The mapping of types becomes:

$$\alpha^\bullet \stackrel{\text{def}}{=} (\alpha^\circ)^\uparrow \quad \mathrm{X}^\circ \stackrel{\text{def}}{=} \mathrm{X}^! \quad (\alpha \Rightarrow \beta)^\circ \stackrel{\text{def}}{=} (\overline{\alpha^\circ}\beta^\bullet)^! \quad (\forall \mathrm{X}.\alpha)^\circ \stackrel{\text{def}}{=} \forall \mathrm{X}.((\alpha^\circ)^\uparrow)^!$$

Write $[\![M:\alpha]\!]_u$ for the encoding of a polymorphic λ-term $M:\alpha$. Then, setting $\cong_{\forall\exists}$ to be the contextual congruence over linear polymorphic processes discussed at the end of Section 4, we obtain:

Theorem 3. (full abstraction) *Let* $\vdash M_{1,2}:\alpha$. *Then* $M_1 \cong_\forall M_2 : \alpha$ *if and only if* $\vdash_{\mathrm{I}} [\![M_1:\alpha]\!]_u \cong_{\forall\exists} [\![M_2:\alpha]\!]_u \triangleright u:\alpha^\circ$.

The proof uses definability arguments based on innocence as in [7] (with additional treatment of contravariant universal types), see [6] for details.

References

[1] Abadi, M., Cardelli, L., and Curien, P.-L. Formal parametric polymorphism. *TCS 121*, 1-2 (1993), 9–58.

[2] Abramsky, S., Honda, K., and McCusker, G. Fully abstract game semantics for general reference. In *LICS'98* (1998), IEEE, pp. 334–344.
[3] Abramsky, S., and Lenisa, M. Axiomatizing fully complete models for ML polymorphic types. In *Proc. of MFCS'2000* (2000).
[4] Abramsky, S., and Lenisa, M. A fully-complete PER model for ML polymorphic types. In *Proc. of CSL'2000*, LNCS. Springer, 2000.
[5] Berger, M. *Towards Abstractions for Distributed Systems.* PhD thesis, Imperial College, Department of Computing, 2002.
[6] Berger, M., Honda, K., and Yoshida, N. Full version of this paper, available at `www.dcs.qmul.ac.uk/~{martinb,kohei}` and `www.doc.ic.ac.uk/~yoshida`.
[7] Berger, M., Honda, K., and Yoshida, N. Sequentiality and the π-calculus. In *Proc. TLCA'01* (2001), no. 2044 in LNCS, Springer, pp. 29–45.
[8] Girard, J.-Y. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur.* PhD thesis, Universite de Paris VII, 1972.
[9] Girard, J.-Y. Linear logic. *Theoretical Computer Science 50* (1987).
[10] Grossman, D. Existential types for imperative languages. In *ESOP02* (2002), LNCS 2305, Springer, pp. 21–35.
[11] Honda, K., and Yoshida, N. Game-theoretic analysis of call-by-value computation. *TCS 221* (1999), 393–456.
[12] Honda, K., and Yoshida, N. A uniform type structure for secure information flow. In *POPL'02: A full version as DOC Report 2002/13, Imperial College,* available at `www.dcs.qmul.ac.uk/~kohei` (2002).
[13] Hughes, D. J. D. Games and definability for system F. In *LICS'97* (1997), IEEE Computer Society Press, pp. 76–86.
[14] Hyland, J. M. E., and Ong, C. H. L. On full abstraction for PCF. *Information and Computation 163* (2000), 285–408.
[15] Lazic, R., Newcomb, T., and Roscoe, A. On model checking data-independent systems with arrays without reset. Tech. Rep. RR-02-02, Oxford University, 2001.
[16] Leroy, X. *Polymorphic typing of an algorithmic language.* PhD thesis, University of Paris, 1992.
[17] Milner, R., Parrow, J., and Walker, D. A calculus of mobile processes, parts I and II. *Info. & Comp. 100*, 1 (1992), 1–77.
[18] Milner, R., Tofte, M., and Harper, R. W. *The Definition of Standard ML.* MIT Press, 1990.
[19] Mitchell, J. C. On the equivalence of data representation. In *Artificial Intelligence and Mathematical Theory of Computation* (1991).
[20] Mitchell, J. C. *Foundations for Programming Languages.* MIT Press, 1996.
[21] Murawski, A., and Ong, C.-H. L. Evolving games and essential nets for affine polymorphism. In *Proc. of TLCA'01* (2001), no. 2044 in LNCS, Springer.
[22] Pierce, B., and Sangiorgi, D. Behavioral equivalence in the polymorphic pi-calculus. *Journal of ACM 47*, 3 (2000), 531–584.
[23] Pierce, B. C., and Turner, D. N. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 2000.
[24] Pitts, A., and Stark, I. Operational reasoning for functions with local state. In *HOOTS'98* (1998), CUP, pp. 227–273.
[25] Pitts, A. M. Existential Types: Logical Relations and Operational Equivalence. In *Proceedings ICALP'98* (1998), no. 1443 in LNCS, Springer, pp. 309–326.
[26] Pitts, A. M. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science 10* (2000), 321–359.

[27] Plotkin, G., and Abadi, M. A logic for parameteric polymorphism. In *LICS'98* (1998), IEEE Press, pp. 42–53.
[28] Reynolds, J. C. Types, abstraction and parametric polymorphism. In *Information Processing 83* (1983), R. E. A. Mason, Ed.
[29] Tofte, M. Type inefrence for polymorphic references. LNCS 2305, Springer, pp. 21–35.
[30] Turner, D. N. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1996.
[31] Vasconcelos, V. Typed concurrent objects. In *Proceedings of ECOOP'94* (1994), LNCS, Springer, pp. 100–117.
[32] Yoshida, N., Berger, M., and Honda, K. Strong Normalisation in the π-Calculus. In *LICS'01* (2001), J. Halpern, Ed., IEEE, pp. 311–322.
[33] Yoshida, N., Honda, K., and Berger, M. Linearity and bisimulation. In *FoSSaCs'02* (2002), vol. 2303 of *LNCS*, Springer, pp. 417–433.

Model Checking Lossy Channels Systems Is Probably Decidable

Nathalie Bertrand and Philippe Schnoebelen

Lab. Spécification & Vérification
ENS de Cachan & CNRS UMR 8643
61, av. Pdt. Wilson, 94235 Cachan Cedex France
{bertrand|phs}@lsv.ens-cachan.fr

Abstract. Lossy channel systems (LCS's) are systems of finite state automata that communicate via unreliable unbounded fifo channels. We propose a new probabilistic model for these systems, where losses of messages are seen as faults occurring with some given probability, and where the internal behavior of the system remains nondeterministic, giving rise to a reactive Markov chains semantics. We then investigate the verification of linear-time properties on this new model.

1 Introduction

Verification of channel systems. Channel systems [BZ83] are systems of finite state automata that communicate via asynchronous unbounded fifo channels. They are a natural model for asynchronous communication protocols, used as the semantical basis of protocol specification languages such as SDL and Estelle. *Lossy channel systems* [Fin94, AJ96b] are a special class of channel systems where messages can be lost while they are in transit, without any notification. These lossy systems are the natural model for fault-tolerant protocols where the communication channels are not supposed to be reliable.

Surprisingly, while channel systems are Turing-powerful [BZ83], several verification problems become decidable when one assumes channels are lossy: reachability, safety properties over traces, inevitability properties over states, and fair termination are decidable for lossy channel systems [Fin94, CFP96, AJ96b, MS02].

This does not mean that lossy channel systems are an artificial model where, since no communication can be fully enforced, everything becomes trivial. To begin with, many important problems are undecidable: recurrent reachability properties are undecidable [AJ96a], so that model checking of liveness properties is undecidable too. Furthermore, boundedness is undecidable [May00], as well as all behavioral equivalences [Sch01]. Finally, none of the decidable problems listed in the previous paragraph can be solved in primitive recursive time [Sch02]!

Probabilistic losses. When modeling real-life protocols, it is natural to see message losses as some kind of faults having a probabilistic behavior. This idea led

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 120–135, 2003.

to the introduction of a Markov chain model for lossy channel systems [PN97]. Essentially the same model allowed Baier and Engelen to show that qualitative model checking is decidable, i.e. it can be decided whether a linear-time property holds *almost surely*, that is, with probability 1 [BE99]. This is a smart way of using randomization to circumvent the undecidability of temporal model checking in the non-probabilistic case. However, this result has several limitations: (l1) it requires that the channel system itself is seen as choosing probabilistically between its transitions, (l2) it assumes that there is a fixed probability p that "the current step is a loss", and (l3) it only gives decidability for $p \geq 0.5$, an unrealistically large value (using a slightly different model, [ABPJ00] shows that decidability is lost if p is not large enough).

Our contribution. We propose an improved approach that addresses the above-mentioned limitations. Our first idea is to use a more realistic probabilistic model for losses, where *any message* has a fixed probability $\tau > 0$ of being lost during the current step, independently of other messages possibly in transit at the same time. We call it the *local-fault model* (and refer to the proposal by [PN97] as the *global-fault model*). In our local-fault model, qualitative model checking is decidable whatever the value of τ (thus our solution to limitation (l2) solves (l3) as well).

Our second idea attacks limitation (l1): we move from Markov chains to *reactive Markov chains* (or, equivalently, *Markovian decision processes*) as the probabilistic model for lossy channel systems: this allows combining a probabilistic behavior for losses with a *nondeterministic* behavior for the channel system. The verification problems we investigate are whether a linear-time property holds almost surely *under any scheduling policy* (the *adversarial* viewpoint). We show that, while the problem is undecidable in general, there exist some decidable subcases (natural subsets of temporal properties). Furthermore, the problem becomes decidable when we restrict ourselves to *finite-memory* scheduling policies only. Finally, it turns out that these verification problems are insensitive to the precise value of the fault rate τ.

Since our decision procedures reduce probabilistic model checking to the kind of reachability questions that have been successfully verified in practice (e.g. [AAB99]), we believe our ideas will provide a nice way of verifying liveness properties on channel systems with probabilistic losses: the approximations "almost surely" and "under any finite-memory scheduling policy" are very reasonable and only retract minimally from the rigid "surely" and "for all scheduling policies" that are the standard goals in algorithmic verification.

Related work. Verifying probabilistic lossy channel systems combines issues from the verification of infinite-state systems and from the verification of probabilistic systems [1]. These two fields are technically quite involved and it seems that, to date, the only joint instance that has been investigated are the probabilistic lossy

[1] Here we do not mean systems where the *timings* are probabilistic like, for example, continuous time Markov chains [BKH99].

systems. We already explained how our work is a continuation of [PN97, BE99, ABPJ00] and depart from these earlier papers. The local-fault model has been independently proposed by Abdulla and Rabinovich [AR03] who proved a result essentially equivalent to our Theorem 5.4 (but did not investigate adversarial verification).

Outline of the paper. Section 2 sets the necessary background on the verification of infinite Markov chains. Channel systems are presented in Section 3, before we discuss probabilistic losses in Section 4 and study probabilistic lossy systems (PLCS's) in Section 5. Nondeterministic PLCS's are defined in Section 6 and their verification is studied in Section 7. For lack of space, many proofs have been omitted in this extended abstract: they can be found in the full version.

2 (Reactive) Markov Chains and Their Verification

We assume some familiarity with Markov chains and only introduce the notations we need in the rest of the paper (we mostly follow [Var99]).

Definition 2.1. *A* Markov chain *is a tuple* $M = \langle W, P, P_0 \rangle$ *of a countable set of* configurations $W = \{\sigma, \ldots\}$*, a* transition probability function $P : W^2 \mapsto [0,1]$ *such that* $\sum_{\sigma' \in W} P(\sigma, \sigma') = 1$ *for all* $\sigma \in W$*, and an* initial probability distribution $P_0 : W \mapsto [0,1]$.

M is *bounded* when there exists $e > 0$ s.t. $P(\sigma, \sigma') > 0$ entails $P(\sigma, \sigma') \geq e$ (i.e. probabilities are not arbitrarily low). M is *finite* when W is. Finite Markov chains are bounded.

A *run* of M is an infinite sequence $\pi \in W^\omega$ of configurations. The set of runs W^ω is turned into a probability space in the standard way: the measure μ of events is first defined on basic cylinders with:

$$\mu(\{\pi \mid \pi \text{ starts with } \sigma_0, \sigma_1, \ldots, \sigma_n\}) \stackrel{\text{def}}{=} P_0(\sigma_0)P(\sigma_0, \sigma_1) \cdots P(\sigma_{n-1}, \sigma_m) \quad (1)$$

and is then extended to the Borel field they generate (see [Var99, Pan01]).

Underlying any Markov chain M is the transition system (the directed graph) G_M where there is a transition $\sigma \to \sigma'$ iff $P(\sigma, \sigma') > 0$. This explains why we often rely on standard graph-theoretic terminology and write statements like "σ is reachable from σ_0", etc., for notions that do not depend on the precise values of the transition probability function. E.g. the measure (1) is non-zero iff σ_0 is a possible initial configuration and $\sigma_0 \to \sigma_1 \to \ldots \to \sigma_n$ is a path in G_M.

2.1 Reactive Markov Chains

Reactive Markov chains [Var99], called "concurrent Markov chains" in [Var85, HSP83], were introduced for modeling systems whose behavior has both probabilistic and nondeterministic aspects. They are a special (and equivalent) form of *Markovian decision processes* [Der70], where the system nondeterministically picks what will be its next step, and the outcome of that step follows some probability law.

Definition 2.2. *A* reactive Markov chain *(a RMC) is a tuple* $M = \langle W, N, P, P_0 \rangle$ *s.t.* $\langle W, P, P_0 \rangle$ *is a Markov chain, and* $N \subseteq W$ *is the subset of* nondeterministic configurations.

The configurations in $W \setminus N$ are called *probabilistic*. For a nondeterministic σ, the exact value of $P(\sigma, \sigma') > 0$ has no importance (apart from being positive): it just means that, when in σ, σ' is a possible next configuration.

The behavior of a RMC $M = \langle W, N, P, P_0 \rangle$ is driven by the nondeterministic choices and the probabilistic behavior. This is formalized by introducing the notion of a *scheduler* (also called *adversary*, or *(scheduling) policy*) that is responsible for the nondeterministic choices. Formally, a scheduler for M is a mapping $u : W^*N \to W$ such that $u(\sigma_0 \dots \sigma_n) = \sigma'$ implies $P(\sigma_n, \sigma') > 0$. The intuition is that, when the system is in a nondeterministic configuration σ_n, u selects a next configuration σ' among the allowed ones, based on the history $\sigma_0 \dots \sigma_n$ of the computation (we do not consider more general notions of adversaries).

Combining a RMC M with a scheduler u gives a *bona fide* Markov chain $M^u = \langle W^+, P^u, P_0^u \rangle$ describing the stochastic behavior of M against u. Intuitively, M^u is obtained by unfolding M into a tree, with W^+ the set of non-empty histories [2], and pruning branches that do not obey u. Formally, for any $x \in W^+$

$$P^u(x\sigma, x\sigma\sigma') \stackrel{\text{def}}{=} \begin{cases} P(\sigma, \sigma') & \text{if } \sigma \notin N, \\ 1 & \text{if } \sigma \in N \text{ and } u(x\sigma) = \sigma', \\ 0 & \text{otherwise,} \end{cases}$$

and $P^u(x\sigma, y\sigma') = 0$ when $y \neq x\sigma$. Finally, P_0^u is like P_0 on histories having length 1, and zero on longer histories. It is readily verified that M^u is indeed a Markov chain.

2.2 Verification for Markov Chains

We address verification of linear-time properties that can be expressed in temporal logic (TL), or second-order monadic logic on runs (MLO), and that do not refer to quantitative information.

Classically such properties can be given under the form of a Büchi automaton that recognizes exactly the correct runs, so that TL model checking reduces to repeated reachability of control states in a product system. This approach does apply to Markov chains if the property is represented by a *deterministic* ω-automaton: then the product system is again a Markov chain.

Since deterministic Büchi automata are not expressive enough for TL or MLO, we shall assume the properties are given by deterministic Street automata. Then, in order to check TL or MLO properties on Markov chains, it is enough to be able to check simpler behavioral properties of the form

[2] When describing the behavior of some M^u, it is customary to leave the histories implicit and only consider their last configuration: this informal way of speaking makes M^u look more like M.

$\alpha = \bigwedge_{i=1}^{n}(\Box\Diamond A_i \Rightarrow \Box\Diamond A'_i)$ where, for $i = 1, \ldots, n$, $A_i, A'_i \subseteq W$ (i.e. α is a Street acceptance condition). A run $\pi = \sigma_0, \sigma_1, \ldots$ satisfies such a condition, written $\pi \models \alpha$, if for all $i = 1, \ldots, n$, either $\sigma_j \in A_i$ for finitely many j, or $\sigma_j \in A'_i$ for infinitely many j. The following is standard:

Theorem 2.3. *Let M be a countable Markov chain and α be a Street acceptance condition. Then $\{\pi \mid \pi \models \alpha\}$ is measurable.*

We let $\mu_M(\alpha)$ denote this measure and say that M satisfies α *with probability* p when $\mu_M(\alpha) = p$. We often consider the probability, written $\mathbb{P}(M, \sigma \models \alpha)$ or $\mu_\sigma(\alpha)$, that a given configuration σ satisfies a property α: this is defined as $\mu_{M'}(\alpha)$ for a Markov chain M' obtained from M by changing the initial distribution.

We say that M satisfies α *almost surely* (resp. *almost never*, *possibly*) when M satisfies α with probability 1 (resp. with probability 0, with probability $p > 0$).

Remark 2.4. These notions are inter-reducible: M satisfies α almost surely iff it satisfies $\neg\alpha$ almost never iff it is not the case that it satisfies $\neg\alpha$ possibly. □

2.3 Verification for Markov Chains with a Finite Attractor

Verifying that a *finite* Markov chain almost surely satisfies a Street property is decidable [CY95, Var99]. However, the techniques involved do not always extend to *infinite* chains, in particular to chains that are not bounded.

It turns out it is possible to extend these techniques to countable Markov chains *where a finite attractor exists*. We now develop these ideas, basically by simply streamlining the techniques of [BE99]. Below we assume a given Markov chain $M = \langle W, P, P_0 \rangle$.

Definition 2.5 (Attractors). *A non-empty set $W_a \subseteq W$ of configurations is an* attractor *when*

$$\mathbb{P}(M, \sigma \models \Box\Diamond W_a) = 1 \text{ for all } \sigma \in W \tag{2}$$

The attractor is finite *when W_a is.*

Assume $W_a \subseteq W$ is a finite attractor. We define $G_M(W_a)$ as the finite directed graph $\langle W_a, \rightsquigarrow \rangle$ where the vertices are the configurations from W_a and where there is an edge $\sigma \rightsquigarrow \sigma'$ iff, in M, σ' is reachable from σ by a non-empty path. Observe that the edges in $G_M(W_a)$ are transitive.

In $G_M(W_a)$, we have the usual graph-theoretic notion of (maximal) strongly connected components (SCC's), denoted $B, B', \ldots$. A trivial SCC is a singleton without the self-loop. These SCC's are ordered by reachability and a minimal SCC (i.e. an SCC B that cannot reach any other SCC) is a *bottom SCC* (a BSCC). Observe that, in $G_M(W_a)$, a BSCC B cannot be trivial: since W_a is an attractor, one of its configurations must be reachable from B.

For a run π in M, we write $\lim_{W_a}(\pi)$ for the sets of configurations from W_a that appear infinitely often in π. Necessarily, if $\lim_{W_a}(\pi) = A$ then the configurations in A are inter-reachable and A is included in some SCC of $G_M(W_a)$.

Lemma 2.6. *If* $\mu_\sigma(\{\pi \mid \lim_{W_a}(\pi) = A\}) > 0$ *then* A *is a BSCC of* $G_M(W_a)$.

Assume the BSSC's of $G_M(W_a)$ are $B_1, \ldots, B_k$. Lemma 2.6 and Eq. (2) entail

$$\mu_\sigma(\{\pi \mid \lim_{W_a}(\pi) = B_1\}) + \cdots + \mu_\sigma(\{\pi \mid \lim_{W_a}(\pi) = B_k\}) = 1. \qquad (3)$$

Therefore, for a BSCC B, $\sigma \in B$ entails $\mu_\sigma(\{\pi \mid \lim_{W_a}(\pi) = B\}) = 1$. Hence $\mu_{\sigma_0}(\{\pi \mid \lim_{W_a}(\pi) = B\}) > 0$ iff B is reachable from σ_0.

It is now possible to reduce the probabilistic verification of Street properties to a finite number of reachability questions:

Proposition 2.7. *Assume* W_a *is a finite attractor of* M. *Then for any* $\sigma \in W$, $\mathbb{P}(M, \sigma \models \bigwedge_{i=1}^{n}(\Box\Diamond A_i \Rightarrow \Box\Diamond A'_i)) > 0$ *iff there exists a BSCC* B *of* $G_M(W_a)$ *such that* $\sigma \xrightarrow{*} B$ *and, for all* $i = 1, \ldots, n$ $B \xrightarrow{*} A_i$ *implies* $B \xrightarrow{*} A'_i$.

2.4 Verification for Reactive Markov Chains

Verifying reactive Markov chains usually assumes an adversarial viewpoint on schedulers. Typical questions are whether, for all schedulers u, M^u satisfies α almost surely (resp. almost never, resp. possibly)? Cooperative viewpoints (asking whether for some u, M^u satisfies α almost surely ...) are possible but less natural in practical verification situations. We consider them since they appear through dualities anyway (Remark 2.4) and since presenting proofs is often easier under the cooperative viewpoint.

Technically, since we still use properties referring to states of W, one defines whether a path in M^u satisfies a property by projecting it from $(W^+)^*$ to W^* in the standard way [Var99].

One sometimes wants to quantify over a restricted set of schedulers, e.g. for checking that M almost surely satisfies α for all *fair* schedulers (assuming some notion of fairness) [HSP83, Var85]. Such a problem can usually be translated into an instance of the general adversarial problem by stating the fairness assumption in the α part.

However, not all restrictions can be transfered in the property to be checked. In particular we shall consider the restriction to *finite-memory* schedulers: this is a convenient way of ruling out infeasible or exaggeratedly malicious schedulers. Several definitions are possible: here we say that u is *finite-memory* if there is a morphism $h : W^* \to H$ that abstract histories from W^* into a finite monoid H and such that $u(\sigma_0 \ldots \sigma_n) = u'(h(\sigma_0, \ldots, \sigma_n), \sigma_n)$ for some $u' : H \times X \to W$. Thus H is the finite memory on which u', the true scheduler, is based. When H is a singleton, u is *memoryless*.

3 Channel Systems

Perfect channel systems. In this paper we adopt the *extended* model of channel systems *where emptiness of channels can be tested for*.[3]

[3] Our *undecidability* proofs do not rely on the extension.

Definition 3.1 (Channel system). *A* channel system *(with m channels) is a tuple $S = \langle Q, C, \Sigma, \Delta, \sigma_0 \rangle$ where*
– *$Q = \{r, s, \ldots\}$ is a finite set of* control locations *(or* control states*),*
– *$C = \{c_1, \ldots, c_m\}$ is a finite set of m* channels*,*
– *$\Sigma = \{a, b, \ldots\}$ is a finite alphabet of* messages*,*
– *$\Delta \subseteq Q \times Act_C \times Q$ is a finite set of* rules*, where $Act_C \stackrel{def}{=} (C \times \{?, !\} \times \Sigma) \cup (C \times \{= \varepsilon ?\})$ is a set of* actions *parameterized by C and Σ,*
– *$\sigma_0 \in Q \times \Sigma^{*C}$ is the* initial configuration *(see below).*

A rule $\delta \in \Delta$ of the form $(s, c, ?, a, r)$ (resp. $(s, c, !, a, r)$) is written "$s \xrightarrow{c?a} r$" (resp. "$s \xrightarrow{c!a} r$") and means that S can move from control location s to r by reading a from (resp. writing a to) channel c. Reading a is only possible if c is not empty and its first available message is a. A rule of the form $(s, c, = \varepsilon ?, r)$ is written "$s \xrightarrow{c=\varepsilon ?} r$" and means that S can move from s to r if channel c is empty.

Formally, the behavior of S is given via a transition system: a *configuration* of S is a pair $\sigma = \langle r, U \rangle$ where $r \in Q$ is a control location and $U \in \Sigma^{*C}$ is a *channel contents*, i.e. a C-indexed vector of Σ-words: for any $c \in C$, $U(c) = u$ means that c contains u. For $s \in Q$ we write $\uparrow s$ for the set $\{s\} \times \Sigma^{*C}$ of all configurations based on s.

The possible moves between configurations are given by the rules of S. For $\sigma, \sigma' \in W$, we write $\sigma \xrightarrow{\delta}_{\text{perf}} \sigma'$ ("perf" is for *perfect steps*) when:

Reads: $\delta \in \Delta$ is some $s \xrightarrow{c?a} r$, σ is some $\langle s, U \rangle$, $U(c)$ is some $a_1 \ldots a_n$ with $a_1 = a$, and $\sigma' = \langle r, U\{c \mapsto a_2 \ldots a_n\} \rangle$ (using the standard notation $U\{c \mapsto u'\}$ for variants).

Writes: $\delta \in \Delta$ is some $s \xrightarrow{c!a} r$, σ is some $\langle s, U \rangle$, $U(c)$ is some $u \in \Sigma^*$, and $\sigma' = \langle r, U\{c \mapsto u.a\} \rangle$.

Tests: $\delta \in \Delta$ is some $s \xrightarrow{c=\varepsilon ?} r$, σ is some $\langle s, U \rangle$, $U(c) = \varepsilon$, and $\sigma' = \langle r, U \rangle$.

Idling: Finally, we have idling steps $\sigma \xrightarrow{0}_{\text{perf}} \sigma$ in any configuration.

We write $En(\sigma)$ for the set of rules *enabled* in configuration σ. We consider idling as a rule and have $0 \in En(\sigma)$ for all σ. For $\delta \in En(\sigma)$, we further write $Succ_\delta(\sigma)$ to denote the (unique) successor configuration σ' obtained by applying δ on σ. We often omit the superscript δ in steps and only write $\sigma \rightarrow_{\text{perf}} \sigma'$.

Remark 3.2. Allowing the idling rule is a definitional detail that smoothes out Definitions 5.1 and 6.1 (deadlocks are ruled out). It also greatly simplifies the technical developments of section 7 (the possibility of idling gives more freedom to scheduling policies). □

Lossy channel systems. In the standard lossiness model, a lossy step is a perfect step possibly preceded and followed by arbitrary message losses. Here we allow losses only *after* the perfect step. This simplifies the construction of the

probabilistic model and does not modify the semantics in any essential way [4] unlike, e.g., the notion of front-lossiness used in [Fin94, CFP96, ABPJ00, Sch01].

Formally, we write $u \sqsubseteq v$ when u is a subword of v, i.e. u can be obtained by erasing any number of letters (possibly zero) from v. When $u \sqsubseteq v$, it will be useful to identify the set $\rho \subseteq \{1, \ldots, |v|\}$ of positions in v where letters have been erased, and we use the notation "$u \sqsubseteq_\rho v$" for that purpose. E.g. $\texttt{aba} \sqsubseteq_{\{1,2,5\}} \texttt{baabba}$. Observe that, in general, $u \sqsubseteq v$ can be explained by several distinct erasures $\rho, \rho', \ldots$

The subword ordering extends to channel contents and to channel systems configurations in the standard way:

$$U \sqsubseteq V \stackrel{\text{def}}{\Leftrightarrow} U(c) \sqsubseteq V(c) \text{ for all } c \in C,$$
$$\langle r, U\rangle \sqsubseteq \langle s, V\rangle \stackrel{\text{def}}{\Leftrightarrow} r = s \text{ and } U \sqsubseteq V.$$

Erasures extend too: we still write $U \sqsubseteq_\rho V$ but now $\rho \subseteq C \times \mathbb{N}$.

It is now possible to define the lossy steps of channel systems: we write $\sigma \xrightarrow{\delta}_{\text{loss}} \sigma'$ when $\sigma' \sqsubseteq \sigma''$ for some σ'' s.t. $\sigma \xrightarrow{\delta}_{\text{perf}} \sigma''$. Perfect steps are a special case of lossy steps (they have $\sigma' = \sigma''$). Below we omit writing explicitly the "loss" subscript for lossy steps, and are simply careful of writing $\rightarrow_{\text{perf}}$ for all perfect steps.

As usual, $\sigma \xrightarrow{*} \sigma'$ denotes that σ' is *reachable* from σ. We write $\sigma \xrightarrow{+} \sigma'$ when σ' is reachable via a non-empty sequence of steps. The *reachability problem for lossy channel systems* is, given S, σ and σ', to say if σ' is reachable from σ in S. It is know that this problem is decidable (even if testing channels for emptiness is allowed) [AJ96b, CFP96, May00].

A set $A \subseteq W$ of configurations is *reachable from* σ if some $\sigma' \in A$ is. This is denoted $\sigma \xrightarrow{*} A$. One can decide whether $\sigma \xrightarrow{*} A$ just by looking at the minimal elements of A. Since $\sqsubseteq$ is a wqo, any $A \subseteq W$ only has a finite number of minimal elements. Therefore it is decidable whether $\sigma \xrightarrow{*} A$.

4 The Local-Fault Model for Probabilistic Losses

Earlier proposals for probabilistic lossy channels assume there is a fixed probability that the next step is the loss of a message [PN97, BE99]. We argued in the introduction that this model is not very realistic. We prefer a viewpoint where the fixed fault rate is associated with every single message. Then, the probability that a given message is lost at the next step is not influenced by the presence or identity of other messages.[5]

[4] The modification only has to do with where we separate a step from its predecessor and successor steps, i.e. with the granularity of the operational semantics.

[5] This agrees with the actual behavior of many lossy fifo links where each message is handled individually by various components (switches, routers, buffers, ...). Admit-

Formally, we assume given a fixed *fault rate* $\tau \in [0,1]$ that describes the probability that any given message will be lost during the next step. From τ, one derives $p_\tau(U,U')$, the probability that channels with contents U will have contents U' after one round of probabilistic losses:

$$p_\tau(U,U') \;=\; \sum_{\rho \text{ s.t. } U' \sqsubseteq_\rho U} \tau^{|\rho|}(1-\tau)^{|U'|}. \tag{4}$$

Then $p_\tau(U,U') > 0$ iff $U' \sqsubseteq U$ (assuming $0 < \tau < 1$), and $\sum_{U'} p_\tau(U,U') = 1$ for any U. It will be convenient to extend this probability distribution from channel contents to configurations with

$$p_\tau(\langle s,U\rangle,\langle r,V\rangle) \stackrel{\text{def}}{=} \begin{cases} p_\tau(U,V) & \text{if } s = r, \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

Example 4.1. Assume we have a single channel c that contains $u = \mathtt{aab}$. Assume $\tau = 0.1$. Then

$$\begin{array}{llll} p_\tau(\mathtt{aab},\varepsilon) = & \tau^3 & = 0.001 & p_\tau(\mathtt{aab},\mathtt{aa}) = \tau(1-\tau)^2 = 0.081 \\ p_\tau(\mathtt{aab},\mathtt{b}) = & \tau^2(1-\tau) & = 0.009 & p_\tau(\mathtt{aab},\mathtt{ab}) = 2\tau(1-\tau)^2 = 0.162 \end{array}$$

$$p_\tau(\mathtt{aab},\mathtt{a}) = 2\tau^2(1-\tau) = 0.018 \qquad p_\tau(\mathtt{aab},\mathtt{aab}) = (1-\tau)^3 = 0.729$$

Observe that $\sum_{u'} p_\tau(u,u') = 1$. The difference between, e.g., $p_\tau(u,\mathtt{a})$ and $p_\tau(u,\mathtt{b})$, comes from the fact that, starting from u, there are two distinct ways of getting $\mathtt{a}$ by losses, while there is only one way of getting $\mathtt{b}$. □

5 Probabilistic Lossy Channel Systems

A *probabilistic lossy channel system* (PLCS) is a tuple $S = \langle Q, C, \Sigma, \Delta, \sigma_0, D\rangle$ s.t. $\langle Q, C, \Sigma, \Delta, \sigma_0\rangle$ is a channel system, and $D : (\Delta \cup \{0\}) \mapsto (0,\infty)$ is a *weight function* of its rules.

Definition 5.1 (Markov chain semantics of PLCS's). *The Markov chain M_S^τ associated with a PLCS S as above, and a fault rate $\tau \in (0,1)$ is $M_S^\tau \stackrel{def}{=} \langle W, P, P_0\rangle$ where W is the set of configurations of S, $P_0(\sigma_0) \stackrel{def}{=} 1$, and where $P(\sigma,\sigma')$, the probability that S moves from σ to σ' in one step, is given by*

$$P(\sigma,\sigma') \stackrel{def}{=} \frac{\displaystyle\sum_{\delta \in En(\sigma)} D(\delta) \times p_\tau(Succ_\delta(\sigma),\sigma')}{\displaystyle\sum_{\delta \in En(\sigma)} D(\delta)}. \tag{6}$$

tedly, there are situations calling for yet other models: e.g. [ABPJ00] assumes losses only occur when a message enters the queue.

It is readily seen that M_S^τ is indeed a Markov chain. It is usually infinite and not bounded.

An important consequence of the local-fault model is that the more messages are in the queue, the more likely some losses will make the number of messages decrease. We formalize this by introducing a partition $W = W_0 + W_1 + \cdots + W_n + \cdots$ of the set of configurations of M_S^τ given by $W_n \stackrel{\text{def}}{=} \{\sigma \in W \mid |\sigma| = n\}$, with $|\langle r, U\rangle| \stackrel{\text{def}}{=} \sum_c |U(c)|$. Then for any S and τ we have

Lemma 5.2. *For all $e > 0$ there is a rank $I \in \mathbb{N}$ s.t. for all $i \geq I$ and $\sigma \in W_i$*

$$\sum\{P(\sigma, \sigma') \mid \sigma' \in W_0 \cup W_1 \cup \cdots \cup W_{i-1}\} > 1 - e. \tag{7}$$

Corollary 5.3. *In M_S^τ, W_0 is a finite attractor.*

Theorem 5.4. (Decidability of model checking for PLCS's) *The problem of checking whether M_S^τ almost-surely (resp. almost-never, resp. possibly) satisfies a Street property α is decidable.*

Proof. Since reachability is decidable for lossy channel systems, the graph $G_{M_S^\tau}(W_0)$ can be built effectively. Since W_0 is a finite attractor, the graph can be used to check whether $\mathbb{P}(M, \sigma_0 \models \alpha) > 0$ by using the criterion provided by Proposition 2.7 (again, using decidability of reachability). Thus it is decidable whether M_S^τ possibly satisfies α. Now, by Remark 2.4, this entails the decidability of checking whether α is satisfied almost surely (resp. almost never). □

Remark 5.5. From this algorithm, we deduce that, for a PLCS S, whether $\mathbb{P}(M_S^\tau, \sigma_0^\tau \models \alpha) = 1$ does not depend on the exact value of the fault rate τ or the weight function D of S. □

6 Lossy Channel Systems as Reactive Markov Chains

Seeing LCS's as Markov chains requires that we see the nondeterministic choice between enabled transitions as being made probabilistically (witness the D weight function in PLCS's).

However, it is more natural to see these choices as being made nondeterministically: this nondeterminism models the lack of any assumption regarding scheduling policies or relative speeds (in concurrent systems), or the lack of any information regarding values that have been abstracted away (in abstract models), or the latitude left for later implementation decisions (in early designs). In all these situations, it is not natural to assume the choices are probabilistic. Even if, for qualitative properties, the exact values in the weight function are not relevant (Remark 5.5), the probabilistic viewpoint enforces a very strong fairness hypothesis on the nondeterministic choices, something which is not suitable except perhaps for concurrent systems.

For these reasons, it is worthwhile to go beyond the Markov chain model and use reactive Markov chains. Below, a *nondeterministic probabilistic lossy channel system* (a NPLCS) is simply a LCS where losses are probabilistic so that the semantics is given under the form of a RMC instead of a transition system.

Definition 6.1. (Reactive Markov chain semantics of NPLCS's) *The RMC associated with a NPLCS S and a fault rate τ is $M_S^\tau = \langle W_+ \cup W_-, W_+, P, P_0\rangle$ where W_+ and W_- are two copies of the set $Q \times \Sigma^{*C}$. P_0 selects $\sigma_{0,+}$, the initial configuration, and P is given by*

$$P(\langle q, U\rangle_+, \langle q', U'\rangle_-) > 0 \text{ iff } \langle q, U\rangle \to_{\text{perf}} \langle q', U'\rangle, \tag{8}$$

$$P(\langle q, U\rangle_-, \langle q', U'\rangle_+) = \begin{cases} p_\tau(U, U') & \text{if } q = q', \\ 0 & \text{otherwise.} \end{cases} \tag{9}$$

Thus positive configurations are nondeterministic and implement perfect steps of S, reaching negative configurations where message losses are probabilistic. Note that the precise value of $P(\sigma_+, \sigma'_-)$ in (8) is not relevant.

Since the probabilistic configurations are only used as some intermediate steps between nondeterministic configurations, it is tempting to omit mentioning them altogether when discussing the behavior of NPLCS's. Indeed, in the next sections, we rarely write configurations with the "$-$" or "$+$" subscript they require: unless explicitly stated, we always refer to the nondeterministic configurations.

7 Model Checking NPLCS's

Model checking for NPLCS's is trickier than model checking PLCS's, and the existence of the finite attractor does not always allow reducing to a decidable finite problem. The decidability results we provide below rely on the finite attractor and downward-closure of reachability sets.

We considered the general case (checking for a Street property) as well as restricted cases where only properties of the form $\Diamond A$ (reachability), $\Box A$ (invariant), $\bigwedge_i \Box\Diamond A_i$ (conjunction of Büchi properties), and $\bigvee_i \Diamond\Box A_i$. Below we adopt the simplifying assumption that all sets A used in properties either are singletons or have the form $\uparrow\{s_1, \ldots, s_k\}$ for a set of control states $s_1, \ldots, s_k$.

We exhibit some decidable cases and some undecidable ones. Most problems are studied under a cooperative "$\exists u? \ldots$" form because this is easier to reason about, but all results are summarized in the adversarial form in Fig. 2 below.

7.1 Some Decidable Problems

We start by consider properties of the simple form $\alpha = \Diamond A$. We say a set $X \subseteq Q$ is *safe for* α if $\langle x, \varepsilon\rangle \xrightarrow{*}_X A$ for all $x \in X$. Here the notation "$\sigma \xrightarrow{*}_X \sigma'$" denotes reachability under the constraint that only control states from X are used (the constraint applies to the endpoints σ and σ' as well). This coincides with reachability in the LCS $S_{|X}$ obtained from S by deleting control states from $Q \setminus X$, and is thus decidable.

Lemma 7.1. *There exists a scheduler u such that $\mathbb{P}(M_S^{\tau,u}, \langle s, \varepsilon\rangle \models \Diamond A) = 1$ iff s belongs to a safe X.*

Corollary 7.2. *It is decidable whether there exists a scheduler u s.t. $\mathbb{P}(M_S^{\tau,u}, \langle s, \varepsilon\rangle \models \Diamond A) = 1$.*

We now consider properties of the special form $\alpha = \bigwedge_{i=1}^{n} \Box\Diamond A_i$. We say $x \in Q$ is *allowed* if for all $i = 1, \ldots, n$, $\langle x, \varepsilon\rangle \xrightarrow{*} A_i$. Otherwise x is *forbidden*. It is decidable whether x is allowed or forbidden.

Lemma 7.3. *Assume all states in S are allowed. Then there exists a scheduler u s.t. $\mathbb{P}(M_S^{\tau,u}\langle s, \varepsilon\rangle \models \alpha) = 1$.*

Lemma 7.4. *Assume x is forbidden and define $S - x$ as the LCS where control state x has been removed. Then the following are equivalent:*
1. There exists a scheduler u s.t. $\mathbb{P}(M_S^{\tau,u}\langle s, \varepsilon\rangle \models \alpha) = 1$.
2. $x \neq s$ and there exists a scheduler u' s.t. $\mathbb{P}(M_{S-x}^{\tau,u'}\langle s, \varepsilon\rangle \models \alpha) = 1$.

Corollary 7.5. *It is decidable whether there exists a scheduler u s.t. $\mathbb{P}(M_S^{\tau,u}, \langle s, \varepsilon\rangle \models \bigwedge_{i=1}^{n} \Box\Diamond A_i) = 1$.*

7.2 An Undecidable Problem

Theorem 7.6. *The problem of checking whether, given a NPLCS S and a Street property α, $\mathbb{P}(M_S^{\tau,u} \models \alpha) = 1$ for all schedulers u, is undecidable.*

The proof is by reduction from the boundedness problem for LCS's, shown undecidable in [May00].

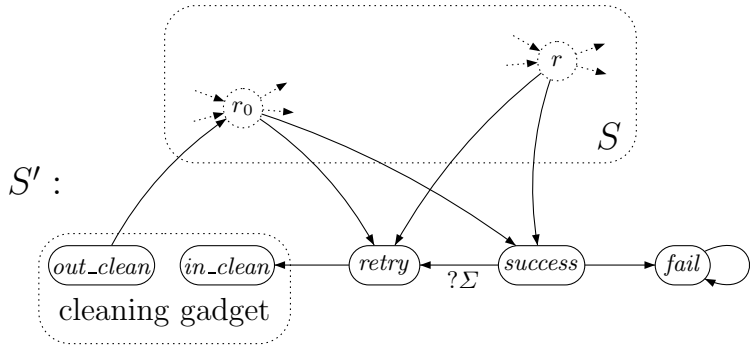


Fig. 1. The NPLCS S' associated with LCS S

Let $S = \langle Q, \{c\}, \Sigma, \Delta, \sigma_0\rangle$ be a single-channel LCS that does not use emptiness tests, and where $\sigma_0 = \langle r_0, \varepsilon\rangle$. We modify S to obtain S', a new LCS. Fig. 1,

where S is in the dashed box, illustrates the construction: S' is obtained by adding three control states *retry*, *success* and *fail*, rules allowing to jump from any S-state $r \in Q$ to *retry* or *success*,[6] and some additional rules between the new states, as depicted in Fig. 1. The "$?\Sigma$" label is a shorthand for all $?a$ where $a \in \Sigma$. We further insert a cleaning gadget (not described) that allows to move from *retry* to r_0 (in a non-blocking way) but empties the channel in the process: this ensures we only jump back to S via its initial configuration $\langle r_0, \varepsilon\rangle$.

If we now see S' as a NPLCS with some fault rate τ, we have

Proposition 7.7. *S is unbounded iff $\mathbb{P}(M_{S'}^{\tau,u}, \sigma_0 \models \Box\Diamond\uparrow success \wedge \Box\Diamond\uparrow retry) > 0$ for some scheduler u.*

Since boundedness of LCS's is undecidable, we obtain Theorem 7.6 even for NPLCS's without emptiness tests.[7]

7.3 More Decidability with Finite-Memory Schedulers!

The scheduler we build in the proof of Proposition 7.7 is not finite-memory. By contrast, all the schedulers exhibited in the proofs in Section 7.1 are finite-memory, so that these decidable problems do not depend on whether we restrict schedulers to the finite-memory ones only.

This observation suggests investigating whether some of our undecidable problems remain undecidable when we restrict to finite-memory schedulers. It turns out this is indeed the case.

We consider a NPLCS S and a Büchi property $\alpha = \bigwedge_{i=1}^{n} \Box\Diamond A_i$. For finite-memory schedulers, cooperative possibly and cooperative almost-sure are related by the following fundamental lemma:

Lemma 7.8. *There exists a finite-memory scheduler u s.t. $\mathbb{P}(M_S^{\tau,u}, \langle s, \varepsilon\rangle \models \alpha) > 0$ iff there is some $s' \in Q$ and a finite-memory scheduler u' s.t $\langle s, \varepsilon\rangle \xrightarrow{*} \langle s', \varepsilon\rangle$ and $\mathbb{P}(M_S^{\tau,u'}, \langle s', \varepsilon\rangle \models \alpha) = 1$.*

Combining with Corollary 7.5, and the decidability of reachability, we obtain:

Corollary 7.9. *It is decidable whether there exists a finite-memory scheduler u s.t. $\mathbb{P}(M_S^{\tau,u}, \langle s, \varepsilon\rangle \models \bigwedge_{i=1}^{n} \Box\Diamond A_i) > 0$.*

Hence the impossibility stated in Theorem 7.6 can be circumvented with:

Theorem 7.10. *The problem of checking whether, given a NPLCS S and a Street property α, $\mathbb{P}(M_S^{\tau,u} \models \alpha) = 1$ for all finite-memory schedulers u, is decidable.*

[6] Via some internal rule where no reading or writing or test takes place. Since such rules are easily simulated by writing to a dummy channel, we simplified Def. 3.1 and omitted them.

[7] Observe that, because of the idling rule, formulae of the form $\Box\Diamond A$ where A is some $\uparrow\{s_1, \ldots, s_n\}$, lead to decidable adversarial problems! This is an artifact and undecidability reappears as soon as we consider slightly more general sets A, e.g. $A \stackrel{\text{def}}{=} \uparrow success \setminus \langle success, \varepsilon\rangle$.

8 Conclusions and Perspectives

When verifying lossy channel systems, adopting a probabilistic view of losses it is a way of enforcing progress and ruling out some unrealistic behaviors (under probabilistic reasoning, it is extremely unlikely that all messages will be lost). Progress could be enforced with fairness assumptions, but assuming fair losses makes verification undecidable [AJ96a, MS02]. It seems this undecidability is an artifact of the standard rigid view asking whether no incorrect behavior exists, when we could be content with the weaker statement that incorrect behaviors are extremely unlikely.

We proposed a model for channel systems where at each step losses of messages occur with some fixed probability $\tau \in (0, 1)$, and where the nondeterministic nature of the channel systems model is preserved. This model is more realistic than earlier proposals since it uses the local-fault model for probabilistic losses, and since it does not require to view the rules of the system as probabilistic. (Picking a meaningful value for τ is not required since the qualitative properties we are interested in do not depend on that value.)

We advocate a specific approach to the verification of these systems: *check that properties hold almost surely under any finite-memory scheduling policy.* It seems this approach is feasible: these adversarial model checking questions can be reduced to the decidable reachability questions that are usually verified on channel systems.

Several questions remain unanswered, and they are good candidates for further work:
1. Fig. 2, summarizing our results on the decidability of adversarial verification *when there is no restriction to finite-memory schedulers*, should be completed. In the table, "D" and "U" stand for decidable and undecidable. Some decidability results are trivial and labeled with "d".
2. Allowing idling makes our decidability proofs much easier (Remark 3.2). We believe this is just a technical simplification that has no impact on decidability, but this should be formally demonstrated.
3. On theoretical grounds, it would be interesting to try to extend our work and consider RMC models of LCS's where the probabilistic states are not limited to message losses but could accommodate probabilistic choices between some rules.

References

[AAB99] P. A. Abdulla, A. Annichini, and A. Bouajjani. Symbolic verification of lossy channel systems: Application to the bounded retransmission protocol. In *Proc. 5th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, vol. 1579 of *Lect. Notes Comp. Sci.*, pages 208–222. Springer, 1999.

[ABPJ00] P. A. Abdulla, C. Baier, S. Purushothaman Iyer, and B. Jonsson. Reasoning about probabilistic lossy channel systems. In *Proc. 11th Int. Conf. Concurrency Theory (CONCUR'2000)*, vol. 1877 of *Lect. Notes in Computer Sci.*, pages 320–333. Springer, 2000.

	$\mathbb{P}(\ldots) = 1$	$\mathbb{P}(\ldots) = 0$	$\mathbb{P}(\ldots) < 1$	$\mathbb{P}(\ldots) > 0$
$\Diamond A$	d	d	D	d
$\Box A$	d	d	d	D
$\bigwedge_i \Box\Diamond A_i$	?	U	D	?
$\bigvee_i \Diamond\Box A_i$	U	?	?	D
$\bigwedge_i(\Box\Diamond A_i \Rightarrow \Box\Diamond A'_i)$	U	U	?	?

Fig. 2. (Un)Decidability of adversarial model checking in the unrestricted case

[AJ96a] P. A. Abdulla and B. Jonsson. Undecidable verification problems for programs with unreliable channels. *Information and Computation*, 130(1):71–90, 1996.

[AJ96b] P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.

[AR03] P. A. Abdulla and A. Rabinovich. Verification of probabilistic systems with faulty communication. In *Proc. FOSSACS'2003 (this volume)*. Springer, 2003.

[BE99] C. Baier and B. Engelen. Establishing qualitative properties for probabilistic lossy channel systems: An algorithmic approach. In *Proc. 5th Int. AMAST Workshop Formal Methods for Real-Time and Probabilistic Systems (ARTS'99)*, vol. 1601 of *Lect. Notes Comp. Sci.*, pages 34–52. Springer, 1999.

[BKH99] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In *Proc. 26th Int. Coll. Automata, Languages, and Programming (ICALP'99)*, vol. 1644 of *Lect. Notes Comp. Sci.*, pages 142–162. Springer, 1999.

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.

[CFP96] G. Cécé, A. Finkel, and S. Purushothaman Iyer. Unreliable channels are easier to verify than perfect channels. *Information and Computation*, 124(1):20–31, 1996.

[CY95] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, 1995.

[Der70] C. Derman. *Finite-State Markovian Decision Processes*. Academic Press, 1970.

[Fin94] A. Finkel. Decidability of the termination problem for completely specificied protocols. *Distributed Computing*, 7(3):129–135, 1994.

[HSP83] S. Hart, M. Sharir, and A. Pnueli. Termination of probabilistic concurrent programs. *ACM Transactions on Programming Languages and Systems*, 5(3):356–380, 1983.

[May00] R. Mayr. Undecidable problems in unreliable computations. In *Proc. 4th Latin American Symposium on Theoretical Informatics (LATIN'2000)*, vol. 1776 of *Lect. Notes Comp. Sci.*, pages 377–386. Springer, 2000.

[MS02] B. Masson and Ph. Schnoebelen. On verifying fair lossy channel systems. In *Proc. 27th Int. Symp. Math. Found. Comp. Sci. (MFCS'2002)*, vol. 2420 of *Lect. Notes Comp. Sci.*, pages 543–555. Springer, 2002.

[Pan01] P. Panangaden. Measure and probability for concurrency theorists. *Theoretical Computer Sci.*, 253(2):287–309, 2001.

[PN97] S. Purushothaman Iyer and M. Narasimha. Probabilistic lossy channel systems. In *Proc. 7th Int. Joint Conf. Theory and Practice of Software Development (TAPSOFT'97)*, vol. 1214 of *Lect. Notes Comp. Sci.*, pages 667–681. Springer, 1997.

[Sch01] Ph. Schnoebelen. Bisimulation and other undecidable equivalences for lossy channel systems. In *Proc. 4th Int. Symp. Theoretical Aspects of Computer Software (TACS'2001)*, vol. 2215 of *Lect. Notes Comp. Sci.*, pages 385–399. Springer, 2001.

[Sch02] Ph. Schnoebelen. Verifying lossy channel systems has nonprimitive recursive complexity. *Information Processing Letters*, 83(5):251–261, 2002.

[Var85] M. Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proc. 26th IEEE Symp. Foundations of Computer Science (FOCS'85)*, pages 327–338, 1985.

[Var99] M. Y. Vardi. Probabilistic linear-time model checking: An overview of the automata-theoretic approach. In *Proc. 5th Int. AMAST Workshop Formal Methods for Real-Time and Probabilistic Systems (ARTS'99)*, vol. 1601 of *Lect. Notes Comp. Sci.*, pages 265–276. Springer, 1999.

Verification of Cryptographic Protocols: Tagging Enforces Termination

Bruno Blanchet[1,2] and Andreas Podelski[2]

[1] Département d'Informatique
École Normale Supérieure, Paris
Bruno.Blanchet@ens.fr
[2] Max-Planck-Institut für Informatik, Saarbrücken
podelski@mpi-sb.mpg.de

Abstract. In experiments with a resolution-based verification method for cryptographic protocols, we could enforce its termination by *tagging*, a syntactic transformation of messages that leaves attack-free executions invariant. In this paper, we generalize the experimental evidence: we prove that the verification method always terminates for tagged protocols.

1 Introduction

The verification of cryptographic protocols is an active research area, see [1–22]. It is important since the design of protocols is error-prone, and testing cannot reveal potential attacks against the protocols. In this paper, we study a verification technique based on Horn clauses and resolution akin to [4,5,24]. We consider a protocol that is executed in the presence of an attacker that can listen to the network, compute, and send messages. The protocol and the attacker are translated into a set of Horn clauses such that: if the fact $\mathsf{att}(M)$ is not derivable from the clauses, then the protocol preserves the secrecy of the message M in every possible execution. The correctness verified is stronger than the one required since the executions possible in the Horn clause model include the ones where a send or receive instruction can be applied more than once in the same session. In practice, the difference between the correctness criteria does not show (no false alarm arised in our experiments).

The verification technique consists of the translation into Horn clauses, followed by the checking of the derivability of facts $\mathsf{att}(M)$ by a resolution-based algorithm. It has the following characteristics.

- It can verify protocols with an unbounded number of sessions.
- It can easily handle a variety of cryptographic primitives, including shared-key and public-key cryptography (encryption and signatures), hash functions, message authentication codes (mac), and even a simple model of Diffie-Hellman key agreements. It can also be used to verify authenticity [5].
- It is efficient (many examples of protocols of the literature are verified in less than 0.1 s; see [5]).

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 136–152, 2003.

The resolution-based verification algorithm has one drawback: it does not terminate in general. In fact, in our experiments, we detected infinite loops during its application to the Needham-Schroeder shared-key protocol [4] and several versions of the Woo-Lam shared-key one-way authentication protocol [5]. It is always possible to modify the algorithm to make it work on those cases and any finite number of other cases, but that will not affect its inherent non-termination property (inherent by the undecidability of the problem that it tries to solve). In this paper, we investigate an alternative: tagging the protocol.

Tagging consists in adding a unique constant to each message. For instance, to encrypt the message m under the key k, we add the tag c_0 to m, so that the encryption becomes $\mathsf{sencrypt}((\mathsf{c}_0, m), k)$. The tagged protocol retains the intended behaviour of the original protocol; i.e., the attack-free executions are the same. Under attacks, it is possibly more secure. Therefore, tagging is a feature of a good protocol design, as explained e.g. in [2]: the receiver of a message uses the tag to identify it unambiguously; thus tagging prevents *type flaws* that occur when a message is taken for another message. (This is formally proved in [16] for a tagging scheme very similar to ours.) Tagging is also motivated by practical issues: the decoding of incoming messages becomes easier. For all these reasons, tags are already present in protocols such as SSH.

In our experiments (including the protocols mentioned above), we obtained termination after tagging the protocol. In this paper, we give the theory behind the experiments: the resolution-based verification algorithm always terminates on tagged protocols. More precisely, on protocols where tags are added to each use of a cryptographic primitive, which may be among: public-key cryptography where keys are atomic, shared-key cryptography (unrestricted), hash functions, and message authentication codes (mac's).

This means that we show termination for a class of protocols that includes many relevant examples.

2 Horn Clauses Representing a Protocol

This section and the next one recapitulate the necessary background on the translation and the algorithm, using material from [4].

Cryptographic protocols can be translated into Horn clauses, either by hand, as explained in [4,24], or automatically, for instance, from a representation of the protocol in an extension of the pi calculus, as in [1].

The terms in the Horn clauses stand for messages. The translation uses one predicate att. The fact $\mathsf{att}(M)$ means that the attacker may have the term M. The fundamental property of this representation is that if $\mathsf{att}(M)$ is not derivable from the clauses, then the protocol preserves the secrecy of M.

The clauses are of two kinds: the clauses in $\mathcal{R}_{\text{Primitives}}$ that depend only on the signature of the cryptographic primitives (they represent computation abilities of the attacker) and the clauses in $\mathcal{R}_{\text{Prot}}$ that one extracts from the protocol itself.

Tuples:
Constructor: tuple $(M_1, \ldots, M_n)$
Destructors: projections $i\mathsf{th}_n((M_1, \ldots, M_n)) \to M_i$
Shared-key encryption:
Constructor: encryption of M under the key N, $\mathsf{sencrypt}(M, N)$
Destructor: decryption $\mathsf{sdecrypt}(\mathsf{sencrypt}(M, N), N) \to M$
Public-key encryption:
Constructors: encryption of M under the public key N, $\mathsf{pencrypt}(M, N)$
public key generation from a secret key N, $\mathsf{pk}(N)$
Destructor: decryption $\mathsf{pdecrypt}(\mathsf{pencrypt}(M, \mathsf{pk}(N)), N) \to M$
Signatures:
Constructor: signature of M with the secret key N, $\mathsf{sign}(M, N)$
Destructors: signature verification $\mathsf{checksignature}(\mathsf{sign}(M, N), \mathsf{pk}(N)) \to M$
message without signature $\mathsf{getmessage}(\mathsf{sign}(M, N)) \to M$
Non-message-revealing signatures:
Constructors: signature of M with the secret key N, $\mathsf{nmrsign}(M, N)$
constant true
Destructor: signature verification $\mathsf{nmrchecksign}(\mathsf{nmrsign}(M, N), \mathsf{pk}(N), M) \to \mathsf{true}$
One-way hash functions:
Constructor: hash function $\mathsf{hash}(M)$.
Message authentication codes, keyed hash functions:
Constructor: mac of M with key N, $\mathsf{mac}(M, N)$

Fig. 1. Constructors and destructors

Attacker Clauses ("$\mathcal{R}_{\text{Primitives}}$") The protocols use cryptographic primitives of two kinds: constructors and destructors (see Figure 1). A constructor f is used to build up a new term $f(M_1, \ldots, M_n)$. For example, the term $\mathsf{sencrypt}(M, N)$ is the encoding of the term M with the key N (by shared-key encryption). A destructor g applied to terms $M_1, \ldots, M_n$ yields a term M built up from subterms of $M_1, \ldots, M_n$. It is defined by a finite set $\mathrm{def}(g)$ of equations written as reduction rules $g(M_1, \ldots, M_n) \to M$ where the terms $M_1, ..., M_n, M$ contain only constructors and variables. For example, the rule $\mathsf{sdecrypt}(\mathsf{sencrypt}(M, N), N) \to M$ models the decoding of the term $\mathsf{sencrypt}(M, N)$ with the same key used for the encoding.

The attacker can form new messages by applying constructors and destructors to already obtained messages. This is modeled, for instance, by the following clauses for shared-key encryption.

$$\mathsf{att}(x) \wedge \mathsf{att}(y) \to \mathsf{att}(\mathsf{sencrypt}(x, y)) \qquad (\mathsf{sencrypt})$$
$$\mathsf{att}(\mathsf{sencrypt}(x, y)) \wedge \mathsf{att}(y) \to \mathsf{att}(x) \qquad (\mathsf{sdecrypt})$$

The first clause expresses that if the attacker has the message x and the shared key y, then he can form the message $\mathsf{sencrypt}(x, y)$. The second clause means that if the attacker has the message $\mathsf{sencrypt}(x, y)$ and the key y, then he can obtain the message x (by applying the destructor $\mathsf{sdecrypt}$ and then using the equality between $\mathsf{sdecrypt}(\mathsf{sencrypt}(x, y), y)$ and x according to the reduction rule for $\mathsf{sdecrypt}$).

We furthermore distinguish between *data* and *cryptographic* constructors and destructors and thus, in total, between four kinds of primitives. The set *DataConstr* of data constructors contains those f that come with a destructor g_i defined by $g_i(f(x_1, \ldots, x_n)) \rightarrow x_i$ for each $i = 1, \ldots, n$; i.e. g_i is used for selecting the argument of f in the i-th position. It is generally sufficient to have only tuples as data constructors (with projections as destructors). All other constructors are said to be cryptographic constructors; they form the set *CryptoConstr*. We collect all clauses like the two example clauses above, for each of the four cases, in the set $\mathcal{R}_{\text{Primitives}}$ of clauses or *rules* defined below.

Definition 1 (Program for Primitives, $\mathcal{R}_{\text{Primitives}}$). *The program for primitives, $\mathcal{R}_{\text{Primitives}}$, is the union of the four sets of Horn clauses corresponding to each of the four cases of cryptographic primitives:*

- $\mathcal{R}_{\text{CryptoConstr}}$ *is the set of clauses* $\mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \rightarrow \mathsf{att}(f(x_1, \ldots, x_n))$ *where f is a cryptographic constructor.*
- $\mathcal{R}_{\text{DataConstr}}$ *is the set of clauses* $\mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \rightarrow \mathsf{att}(f(x_1, \ldots, x_n))$ *where f is a data constructor.*
- $\mathcal{R}_{\text{CryptoDestr}}$ *is the set of clauses* $\mathsf{att}(M_1) \wedge \ldots \wedge \mathsf{att}(M_n) \rightarrow \mathsf{att}(M)$ *where g is a cryptographic destructor with the reduction rule $g(M_1, \ldots, M_n) \rightarrow M$.*
- $\mathcal{R}_{\text{DataDestr}}$ *is the set of clauses* $\mathsf{att}(f(x_1, \ldots, x_n)) \rightarrow \mathsf{att}(x_i)$ *where f is a data constructor and $i = 1, \ldots, n$.*

Protocol Clauses ("$\mathcal{R}_{\text{Prot}}$") We note $\mathcal{R}_{\text{Prot}}$ the set of *protocol clauses*. These include clauses that directly correspond to send and receive instructions of the protocol and clauses translating the initial knowledge of the attacker.

In a protocol clause of the form

$$\mathsf{att}(M_1) \wedge \ldots \wedge \mathsf{att}(M_n) \rightarrow \mathsf{att}(M)$$

the term M in the conclusion represents the sent message. The hypotheses correspond to messages received by the same host before sending M. Indeed, the clause means that if the attacker has $M_1, ..., M_n$, he can send these messages to a participant who is then going to reply with M, and the attacker can then intercept this message.

If the initial knowledge of the attacker consists of the set of terms S_{Init} (containing e.g. public keys, host names, and a name N that represents all names that the attacker creates), then it is represented by the facts $\mathsf{att}(M)$ where M is a term in S_{Init}.

We explain protocol clauses on the example of the Yahalom protocol [8]:

Message 1.	$A \rightarrow B:$	(A, N_a)
Message 2.	$B \rightarrow S:$	$(B, \{A, N_a, N_b\}_{K_{bs}})$
Message 3.	$S \rightarrow A:$	$(\{B, K_{ab}, N_a, N_b\}_{K_{as}}, \{A, K_{ab}\}_{K_{bs}})$
Message 4.	$A \rightarrow B:$	$(\{A, K_{ab}\}_{K_{bs}}, \{N_b\}_{K_{ab}})$

In this protocol, two participants A and B wish to establish a session key K_{ab}, with the help of a trusted server S. Initially, A has a shared key K_{as} to communicate with S, and B has a shared key K_{bs} to communicate with S. In the

first message, A sends to B his name A and a nonce (fresh value) N_a. Then B creates a nonce N_b and sends to the server his own name B and the encryption $\{A, N_a, N_b\}_{K_{bs}}$ of A, N_a, N_b under the shared key K_{bs}. The server then creates the new (fresh) session key K_{ab}, and sends two encrypted messages to A. The first one $\{B, K_{ab}, N_a, N_b\}_{K_{as}}$ gives the key K_{ab} to A, together with B's name and the nonces (so that A knows that the key is intended to communicate with B). The second message cannot be decrypted by A, so A forwards it to B (message 4). B then obtains the session key K_{ab}. The second part of message 4, $\{N_b\}_{K_{ab}}$, is used to check that A and B really use the same key K_{ab}: B is going to check that he can decrypt the message with the newly received key. We encode only one principal playing each role, since others can be included in the attacker.

Message 1 is represented by the clause

$$\mathsf{att}((\mathsf{host}(\mathsf{Kas}), \mathsf{Na})) \tag{Msg1}$$

meaning that the attacker gets $\mathsf{host}(\mathsf{Kas})$ and Na when intercepting message 1. In this clause, the host name A is represented by $\mathsf{host}(\mathsf{Kas})$. Indeed, the server has a table of pairs (host name, shared key to communicate between that host and the server), and this table can be conveniently represented by a constructor host. This constructor takes as parameter the secret key and returns the host name. So host names are written $\mathsf{host}(k)$. The server can also match a term $\mathsf{host}(k)$ to find back the secret key. The attacker cannot do this operation (he does not have the key table), so there is no destructor clause for host. There is a constructor clause, since the attacker can build new hosts with new host keys:

$$\mathsf{att}(k) \rightarrow \mathsf{att}(\mathsf{host}(k)) \tag{host}$$

Message 2 is represented by the clause:

$$\mathsf{att}((a, na)) \rightarrow \mathsf{att}((\mathsf{host}(\mathsf{Kbs}), \mathsf{sencrypt}((a, na, \mathsf{Nb}(a, na)), \mathsf{Kbs}))) \tag{Msg2}$$

The hypothesis means that a message (a, na) (corresponding to message 1) must be received before sending message 2. It corresponds to the situation in which the attacker sends (a, na) to B, B takes that for message 1, and replies with message 2, which is intercepted by the attacker. (a and na are variables since B accepts any term instead of $\mathsf{host}(\mathsf{Kas})$ and Na.) The nonce N_b is represented by the function $\mathsf{Nb}(a, na)$. Indeed, since a new name is created at each execution, names created after receiving different messages are different. This is modeled by considering names as functions of the messages previously received. This modeling is slightly weaker than creating a new name at each run of the protocol, but it is correct: if a secrecy property is proved in this model, then it is true [1]. The introduced function symbols will be called "name function symbols". (In message 1, the fresh name Na is a constant because there are no previous messages on which it would depend.)

Message 3 is represented by the clause:

$$\begin{aligned}&\mathsf{att}((\mathsf{host}(kbs), \mathsf{sencrypt}((\mathsf{host}(kas), na, nb), kbs)))\\ &\quad\rightarrow \mathsf{att}((\mathsf{sencrypt}((\mathsf{host}(kbs), \mathsf{Kab}(kas, kbs, na, nb), na, nb), kas),\\ &\qquad\quad \mathsf{sencrypt}((\mathsf{host}(kas), \mathsf{Kab}(kas, kbs, na, nb)), kbs)))\end{aligned} \tag{Msg3}$$

using the same principles. At last, message 4 is represented by

$$\mathsf{att}((\mathsf{sencrypt}((b, k, \mathsf{Na}, nb), \mathsf{Kas}), mb)) \rightarrow \mathsf{att}((mb, \mathsf{sencrypt}(nb, k))) \qquad \text{(Msg4)}$$

The message $\mathsf{sencrypt}((\mathsf{host}(\mathsf{Kas}), k), \mathsf{Kbs})$ cannot be decrypted and checked by A, so it is a variable mb.

The goal of the protocol is to establish a secret shared key K_{ab} between A and B. If the key was a constant, say $\mathsf{K_{ab}}$, then the non-derivability of the fact $\mathsf{att}(\mathsf{K_{ab}})$ from the Horn clauses presented so far would prove its secrecy. However, K_{ab}, as received by A, is a variable k. We therefore use the following fact. The key K_{ab} received by A is secret if and only if some constant $\mathsf{secretA}$ remains secret when A sends it encrypted under the key K_{ab}. Thus, we add a clause that corresponds to the translation of an extra message of the protocol, Message 5. $A \rightarrow B : \{\mathsf{secretA}\}_{K_{ab}}$.

$$\begin{aligned} &\mathsf{att}((\mathsf{sencrypt}((\mathsf{host}(\mathsf{Kbs}), k, \mathsf{Na}, nb), \mathsf{Kas}), mb)) \\ &\quad \rightarrow \mathsf{att}(\mathsf{sencrypt}(\mathsf{secretA}, k)) \end{aligned} \qquad \text{(Msg5)}$$

Now, the secrecy of the key K_{ab} received by A can be proved from the non-derivability of the fact $\mathsf{att}(\mathsf{secretA})$ from the set of clauses $\mathcal{R}_{\text{Primitives}} \cup \mathcal{R}_{\text{Prot}}$.

For the Yahalom protocol, the translation yields the union of the following sets of Horn clauses. $\mathcal{R}_{\text{CryptoConstr}}$ contains ($\mathsf{sencrypt}$) and (host), $\mathcal{R}_{\text{CryptoDestr}}$ contains ($\mathsf{sdecrypt}$), $\mathcal{R}_{\text{DataConstr}}$ contains the tuple construction and $\mathcal{R}_{\text{DataDestr}}$ the tuple projections (both not listed), and $\mathcal{R}_{\text{Prot}}$ contains (Msg1), (Msg2), (Msg3), (Msg4) and (Msg5) and three clauses translating the initial knowledge, $\mathsf{att}(\mathsf{N})$, $\mathsf{att}(\mathsf{host}(\mathsf{Kas}))$, and $\mathsf{att}(\mathsf{host}(\mathsf{Kbs}))$.

3 The Resolution-Based Verification Algorithm

To determine whether a fact is derivable from the clauses, we use a resolution-based algorithm explained below. (We use the meta-variables R, H, C, F for rule, hypothesis, conclusion, fact, respectively.)

The algorithm infers new clauses by resolution as follows: From two clauses $R = H \rightarrow C$ and $R' = F \wedge H' \rightarrow C'$ (where F is any hypothesis of R'), it infers $R \circ_F R' = \sigma H \wedge \sigma H' \rightarrow \sigma C'$, where C and F are unifiable and σ is the most general unifier of C and F. The clause $R \circ_F R'$ is the combination of R and R', where R proves the hypothesis F of R'. The resolution is guided by a selection function sel. Namely, $sel(R)$ returns a subset of the hypotheses of R, and the resolution step above is performed only when $sel(R) = \emptyset$ and $F \in sel(R')$.

We can use several selection functions. In this paper, we use:

$$sel(H \rightarrow C) = \begin{cases} \emptyset \text{ if all elements of } H \text{ are of the form } \mathsf{att}(x),\ x \text{ variable} \\ \{F\} \text{ where } F \neq \mathsf{att}(x),\ F \in H, \text{ otherwise} \end{cases}$$

The algorithm uses the following optimizations:

- Decomposition of data constructors: *decomp* takes a clause and returns a set of clauses, built as follows. For each data constructor f, *decomp* replaces recursively all facts $\mathsf{att}(f(M_1, \ldots, M_n))$ with $\mathsf{att}(M_1) \wedge \ldots \wedge \mathsf{att}(M_n)$. When such a fact is in the conclusion of a clause, n clauses are created, with the same hypotheses and the conclusions $\mathsf{att}(M_1), \ldots, \mathsf{att}(M_n)$ respectively. With decomposition, the standard clauses for data constructors and projections can be removed. The soundness of this operation follows from the equivalence between $\mathsf{att}(f(M_1, \ldots, M_n))$ and $\mathsf{att}(M_1) \wedge \ldots \wedge \mathsf{att}(M_n)$ in the presence of the clauses $\mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \rightarrow \mathsf{att}(f(x_1, \ldots, x_n))$ and $\mathsf{att}(f(x_1, \ldots, x_n)) \rightarrow \mathsf{att}(x_i)$ in $\mathcal{R}_{\mathrm{DataConstr}}$ and $\mathcal{R}_{\mathrm{DataDestr}}$.
- Elimination of duplicate hypotheses: *elimdup* takes a clause and returns the same clause after keeping only one copy of duplicate hypotheses.
- Elimination of hypotheses $\mathsf{att}(x)$: *elimattx* eliminates hypotheses $\mathsf{att}(x)$ when x does not appear elsewhere in the clause. Indeed, these hypotheses are always true, since the attacker has at least one term.
- Elimination of tautologies: *elimtaut* eliminates all tautologies (that is, clauses whose conclusion is already in the hypotheses) from a set of clauses.
- *simplify* groups all these simplifications. We extend *elimdup* and *elimattx* naturally to sets of clauses, and define *simplify* = *elimtaut* $\circ$ *elimattx* $\circ$ *elimdup* $\circ$ *decomp*.
- *condense*($\mathcal{R}$) applies *simplify* to each clause in $\mathcal{R}$ and then eliminates subsumed clauses. We say that $H_1 \rightarrow C_1$ subsumes $H_2 \rightarrow C_2$ if and only if there exists a substitution σ such that $\sigma C_1 = C_2$ and $\sigma H_1 \subseteq H_2$. If $\mathcal{R}$ contains clauses R and R', such that R subsumes R', R' is eliminated. (In that case, R can do all derivations that R' can do.)

We now define the algorithm *saturate*($\mathcal{R}_0$). Starting from *condense*($\mathcal{R}_0$), the algorithm adds clauses inferred by resolution with the selection function *sel* and condenses the set of clauses at each iteration step until a fixpoint is reached. When a fixpoint is reached, *saturate*($\mathcal{R}_0$) consists of the clauses R in the fixpoint such that $sel(R) = \emptyset$. By adapting the proof of [4] to this algorithm, it is easy to show that, for any $\mathcal{R}_0$ and any closed fact F, F is derivable from $\mathcal{R}_{\mathrm{All}} = \mathcal{R}_0 \cup \mathcal{R}_{\mathrm{DataConstr}} \cup \mathcal{R}_{\mathrm{DataDestr}}$ if and only if it is derivable from *saturate*($\mathcal{R}_0$) $\cup$ $\mathcal{R}_{\mathrm{DataConstr}}$.

Once the clauses of *saturate*($\mathcal{R}_0$) have been computed, we use a standard backward depth-first search to see if a fact can be derived from *saturate*($\mathcal{R}_0$) $\cup$ $\mathcal{R}_{\mathrm{DataConstr}}$. Taking $\mathcal{R}_0 = \mathcal{R}_{\mathrm{CryptoConstr}} \cup \mathcal{R}_{\mathrm{CryptoDestr}} \cup \mathcal{R}_{\mathrm{Prot}}$, if $\mathsf{att}(M)$ cannot be derived from *saturate*($\mathcal{R}_0$) $\cup$ $\mathcal{R}_{\mathrm{DataConstr}}$ then the protocol preserves the secrecy of M.

The optimizations enable us to weaken the conditions that guarantee termination. For instance, the decomposition of data constructors makes it possible to obtain termination without tagging each data constructor application, while other constructors such as encryption must be tagged. In the Yahalom protocol, for example, without decomposition of data constructors, the algorithm would resolve the clause (Msg2) with itself, immediately yielding an infinite loop.

Another consequence of the optimizations is that not all terms in a clause can be variables. Indeed, when $x \in \{x_1, \ldots, x_n\}$, the clause $\mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \rightarrow \mathsf{att}(x)$ is eliminated since it is a tautology. When $x \notin \{x_1, \ldots, x_n\}$, all hypotheses are eliminated, so the clause becomes $\mathsf{att}(x)$ and all other clauses are eliminated since they are subsumed by $\mathsf{att}(x)$, so the algorithm stops immediately: all facts can be derived. Thus, when $sel(R) = \emptyset$, the conclusion of R is not of the form $\mathsf{att}(x)$. Therefore, the above selection function prevents resolution steps in which $\mathsf{att}(x)$ is unified with another fact (actually, with any other fact, which can lead to non-termination).

4 Sufficient Conditions for Termination

We are now collecting the formal properties of sets of Horn clauses (logic programs, or programs for short) that together entail termination. The properties for *protocol programs* hold for the translation of every protocol. The properties for *plain* protocol programs hold for the translation of protocols with a restriction on their cryptographic primitives and on their keys (this restriction is satisfied by many interesting protocols, including Yahalom for example). The properties for *tagged* protocol programs hold for the translation of those protocols after they have been tagged. The derivability problem for plain protocol programs is undecidable (as can be easily seen by a reduction to two-counter machines). The restriction to tagged programs makes the problem decidable, as will follow.

Given a clause R of the form $\mathsf{att}(M_1) \wedge \ldots \wedge \mathsf{att}(M_n) \rightarrow \mathsf{att}(M_0)$, we say that the terms $M_0, M_1, \ldots, M_n$ are the *terms of* R, and we denote the set of terms of R by $terms(R)$.

Definition 2 (Protocol program). *A protocol program is a set of clauses* $\mathcal{R}_{\mathrm{All}} = \mathcal{R}_{\mathrm{Primitives}} \cup \mathcal{R}_{\mathrm{Prot}}$ *(where* $\mathcal{R}_{\mathrm{Primitives}}$ *is a program for primitives) that comes with a finite set of closed terms* S_0 *such that:*

- *C1. For all clauses* R *in* $\mathcal{R}_{\mathrm{Prot}}$*, there exists a substitution* σ *such that* $terms(\sigma R) \subseteq S_0$*.*
- *C2. Every two subterms of terms in* S_0 *of the form* $a(\ldots)$ *with the same name function symbol* a *are identical.*
- *C3. The second argument of* $\mathsf{pencrypt}$ *in* S_0 *is of the form* $\mathsf{pk}(M)$ *for some* M*.*

The terminology "argument of f in S_0" refers to a term M such that $f(\ldots, M, \ldots)$ is a subterm of a term in S_0. To see why these conditions are satisfied by a translation of a protocol, let us consider the intended messages of the protocol. These are the exchanged messages when the attacker does not intervene and when there is no unexpected interaction between sessions of the protocol. We denote by $M_1, \ldots, M_k$ the closed terms corresponding to these messages. Each participant does not necessarily have a full view of the messages he receives; instead, he accepts all messages that are instances of patterns representing the information he can check. The terms $M_1, \ldots, M_k$ are particular instances

of these patterns. So the protocol is represented by clauses R such that there exists σ such that $terms(\sigma R) \subseteq \{M_1, \ldots, M_k\}$. Defining $S_0 = \{M_1, \ldots, M_k\} \cup S_{\mathrm{Init}}$, we obtain C1.

For instance, the intended messages for the Yahalom protocol are

$M_1 = (\mathsf{host}(\mathsf{Kas}), \mathsf{Na})$
$M_2 = (\mathsf{host}(\mathsf{Kbs}), \mathsf{sencrypt}((\mathsf{host}(\mathsf{Kas}), \mathsf{Na}, M_{N_b}), \mathsf{Kbs}))$
$M_3 = (\mathsf{sencrypt}((\mathsf{host}(\mathsf{Kbs}), M_K, \mathsf{Na}, M_{N_b}), \mathsf{Kas}), \mathsf{sencrypt}((\mathsf{host}(\mathsf{Kas}), M_K), \mathsf{Kbs}))$
$M_4 = (\mathsf{sencrypt}((\mathsf{host}(\mathsf{Kas}), M_K), \mathsf{Kbs}), \mathsf{sencrypt}(M_{N_b}, M_K))$
$M_5 = \mathsf{sencrypt}(\mathsf{secretA}, M_K)$

with $M_{N_b} = \mathsf{Nb}(\mathsf{host}(\mathsf{Kas}), \mathsf{Na})$ and $M_K = \mathsf{Kab}(\mathsf{Kas}, \mathsf{Kbs}, \mathsf{Na}, M_{N_b})$. It is easy to check that the clauses (Msg1)–(Msg5) satisfy the condition C1.

Condition C2 models that each name function symbol is created at a unique occurrence in the protocol. Condition C3 means that, in its intended behaviour, the protocol uses public-key encryption only with public keys.

Definition 3 (Plain protocol program). *A plain protocol program is a protocol program $\mathcal{R}_{\mathrm{All}}$ with associated set of closed terms S_0, such that:*

C4. The only constructors and destructors are those of Figure 1, plus host.
C5. The arguments of pk *and* host *in S_0 are atomic constants.*

Condition C5 essentially means that the protocol only uses pairs of atomic keys for public key cryptography, and atomic keys for long-term secret keys.

Tagging a protocol is a simple syntactic annotation of messages. We add a tag to each application of a primitive sencrypt, pencrypt, sign, nmrsign, hash, mac, such that two applications of the same primitive with the same tag have the same parameters. For example, after tagging the Yahalom protocol becomes:

Message 1. $A \rightarrow B : (A, N_a)$
Message 2. $B \rightarrow S : (B, \{\mathsf{c}_1, A, N_a, N_b\}_{K_{bs}})$
Message 3. $S \rightarrow A : (\{\mathsf{c}_2, B, K_{ab}, N_a, N_b\}_{K_{as}}, \{\mathsf{c}_3, A, K_{ab}\}_{K_{bs}})$
Message 4. $A \rightarrow B : (\{\mathsf{c}_3, A, K_{ab}\}_{K_{bs}}, \{\mathsf{c}_4, N_b\}_{K_{ab}})$

If the original protocol translates to a plain protocol program, its tagged version translates to a tagged protocol program, as defined below.

Definition 4 (Tagged protocol program). *A tagged protocol program is a plain protocol program $\mathcal{R}_{\mathrm{All}}$ with associated set of closed terms S_0 such that:*

*C6. If $f \in \{$*sencrypt, pencrypt, sign, nmrsign, hash, mac*$\}$ occurs in a term in S_0 or in $terms(R)$ for $R \in \mathcal{R}_{\mathrm{Prot}}$, then its first argument is the tuple $(\mathsf{c}, M_1, \ldots, M_n)$ for some constant* c *and terms $M_1, \ldots, M_n$.*
*C7. Every two subterms of terms in S_0 of the form $f((\mathsf{c}, \ldots), \ldots)$ with the same primitive $f \in \{$*sencrypt, pencrypt, sign, nmrsign, hash, mac*$\}$ and the same tag* c *are identical.*

The condition that constant tags appear in $terms(R)$ (Condition C6) means that honest protocol participants always check the tags of received messages (something that the informal description of a tagged protocol leaves implicit) and send tagged terms. The condition also expresses that the initial knowledge of the attacker consists of tagged terms.

5 Termination Proof

Instead of giving the termination proof in one big step, we first consider a special case (Section 5.1), and then describe the modification of the first proof that yields the proof for the general case (Section 5.2).

The special case is defined in terms of the sets $Params_{\mathsf{pk}}$ and $Params_{\mathsf{host}}$ of arguments of pk resp. host in S_0, namely by the condition that these sets each have at most one element.

This restriction is meaningful in terms of models of protocols: it corresponds to merging several keys. In the example of the Yahalom protocol, this means that, in the clauses, the keys Kas and Kbs should be replaced with a single key, k_0 (so the host names $A = \mathsf{host}(\mathsf{Kas})$ and $B = \mathsf{host}(\mathsf{Kbs})$ are replaced with a single name $\mathsf{host}(\mathsf{k}_0)$). When studying secrecy, merging all keys of honest hosts in this way helps to model cases in which one host plays several roles in the protocol. The secrecy for the clauses with merged keys implies secrecy for the protocol without merged keys. However, this merging is not acceptable for authenticity [5]. This is why we also consider the general case in Section 5.2.

5.1 The Special Case of One Key

We now define weakly tagged programs by the conditions that we use in the first termination proof. In the special case, these conditions are strictly more general than tagged protocol programs. This plays a role to deduce termination for protocols that are not explicitly tagged (see Remark 1).

A term is said to be *non-data* when it is not of the form $f(\ldots)$ with f in $DataConstr$. The set $sub(S)$ contains the subterms of terms in the set S.

The set $tagGen$ contains the non-variable non-data subterms of terms of clauses in $\mathcal{R}_{\mathrm{Prot}}$ and of terms $M_1, \ldots, M_n$ in clauses of the form $\mathsf{att}(f(M_1, \ldots, M_n)) \wedge \mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_m) \rightarrow \mathsf{att}(x)$ in $condense(\mathcal{R}_{\mathrm{CryptoDestr}})$ (this is the form required in W1 below). This set summarizes the terms that appear in the clauses and that should be tagged.

Definition 5 (Weakly tagged programs). *A program $\mathcal{R}_{\mathrm{All}}$ of the form $\mathcal{R}_{\mathrm{All}} = \mathcal{R}_{\mathrm{Primitives}} \cup \mathcal{R}_{\mathrm{Prot}}$ (where $\mathcal{R}_{\mathrm{Primitives}}$ is a program for primitives) is weakly tagged if there exists a finite set of closed terms S_0 such that:*

W1. All clauses in the set $\mathcal{R}'_{\mathrm{CryptoDestr}} = condense(\mathcal{R}_{\mathrm{CryptoDestr}})$ are of the form

$$\mathsf{att}(f(M_1, \ldots, M_n)) \wedge \mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_m) \rightarrow \mathsf{att}(x)$$

where $f \in CryptoConstr$, x is one of $M_1, \ldots, M_n$, and $f(M_1, \ldots, M_n)$ is more general than every term of the form $f(\ldots)$ in $sub(S_0)$.

W2. For all clauses R in $\mathcal{R}_{\text{Prot}}$, there exists a substitution σ such that $terms(\sigma R) \subseteq S_0$.

W3. If two terms M_1 and M_2 in tagGen unify, N_1 is an instance of M_1 in $sub(S_0)$, and N_2 is an instance of M_2 in $sub(S_0)$, then $N_1 = N_2$.

Condition W3 is the key of the termination proof. We are going to show the following invariant: all terms in the generated clauses are instances of terms in *tagGen* and have instances in $sub(S_0)$. This condition makes it possible to prove that, when unifying two terms satisfying the invariant, the result of the unification also satisfies the invariant; this is because the instances in $sub(S_0)$ of those two terms are in fact equal. Condition W1 guarantees that this continues to hold if only one of the two terms satisfies the invariant and the other stems from a clause in $\mathcal{R}'_{\text{CryptoDestr}}$.

Proposition 1. *A tagged protocol program where $Params_{\mathsf{host}}$ and $Params_{\mathsf{pk}}$ each have at most one element, is weakly tagged.*

Proof. For condition W1, the clauses for sdecrypt, pdecrypt, and getmessage are:

$$\mathsf{att}(\mathsf{sencrypt}(x, y)) \wedge \mathsf{att}(y) \rightarrow \mathsf{att}(x) \qquad \text{(sdecrypt)}$$
$$\mathsf{att}(\mathsf{pencrypt}(x, \mathsf{pk}(y))) \wedge \mathsf{att}(y) \rightarrow \mathsf{att}(x) \qquad \text{(pdecrypt)}$$
$$\mathsf{att}(\mathsf{sign}(x, y)) \rightarrow \mathsf{att}(x) \qquad \text{(getmessage)}$$

and they satisfy condition W1 provided that all public-key encryptions in S_0 are of the form $\mathsf{pencrypt}(M_1, \mathsf{pk}(M_2))$ (that is C3). The clauses for checksignature and nmrchecksign are

$$\mathsf{att}(\mathsf{sign}(x, y)) \wedge \mathsf{att}(\mathsf{pk}(y)) \rightarrow \mathsf{att}(x) \qquad \text{(checksignature)}$$
$$\mathsf{att}(\mathsf{nmrsign}(x, y)) \wedge \mathsf{att}(\mathsf{pk}(y)) \wedge \mathsf{att}(x) \rightarrow \mathsf{att}(\mathsf{true}) \qquad \text{(nmrchecksign)}$$

These two clauses are subsumed respectively by the clauses for getmessage (given above) and true (which is simply att(true) since true is a zero-ary constructor), so they are eliminated by *condense*, i.e., they are not in $\mathcal{R}'_{\text{CryptoDestr}}$. (This is important, because they do not satisfy condition W1.)

Condition W2 is identical to condition C1. We now prove condition W3. Let

$$\begin{aligned} S_1 = \{ & f((\mathsf{c}_i, x_1, \ldots, x_n), x'_2, \ldots, x'_{n'}) \mid \\ & f \in \{\mathsf{sencrypt}, \mathsf{pencrypt}, \mathsf{sign}, \mathsf{nmrsign}, \mathsf{hash}, \mathsf{mac}\}\} \\ & \cup \{a(x_1, \ldots, x_n) \mid a \text{ name function symbol}\} \\ & \cup \{\mathsf{pk}(x), \mathsf{host}(x)\} \cup \{c \mid c \text{ atomic constant}\} \end{aligned}$$

By condition C4, the only term in *tagGen* that comes from clauses of $\mathcal{R}'_{\text{CryptoDestr}}$ is $\mathsf{pk}(x)$. Using condition C6, all terms in *tagGen* are instances of terms in S_1 (noticing that *tagGen* does not contain variables). Using conditions C2, C5, C7, and the fact that $Params_{\mathsf{pk}}$ and $Params_{\mathsf{host}}$ have at most one element, each term in S_1 has at most one instance in $sub(S_0)$.

If M_1 and M_2 in *tagGen* unify, they are both instances of the same element M' in S_1 (since different elements of S_1 do not unify with each other). Let N_1 and N_2 be any instances of M_1 and M_2 (respectively) in $sub(S_0)$. Then N_1 and N_2 are instances of $M' \in S_1$ in $sub(S_0)$ so $N_1 = N_2$. Thus we obtain W3. □

Remark 1. The Yahalom protocol is in fact weakly tagged without explicitly adding constant tags (after merging the keys Kas and Kbs). Indeed, since different encryptions in the protocol have a different arity, we can take $\mathsf{sencrypt}((x_1, \ldots, x_n), x')$ in S_1 in the proof above, and use the same reasoning as above to prove the condition W3. This shows both that type flaws cannot happen in the original protocol, and that the algorithm also terminates on the original protocol. We can say that the protocol is "implicitly tagged": the arity replaces the tag. This situation happens in some other examples, and can partly explain why the algorithm often terminates even for protocols without explicit tags.

A term is *top-tagged* when it is an instance of a term in *tagGen*. Intuitively, referring to the case of explicit constant tags, top-tagged terms are terms whose top function symbol is tagged. A term is *fully tagged* when all its non-variable non-data subterms are top-tagged.

We next show the invariant that all terms in the generated clauses are non-data, fully tagged, and have instances in $sub(S_0)$. Using this invariant, we show that the size of an instance in $sub(S_0)$ of a clause obtained by resolution from R and R' is smaller than the size of an instance of R or R' in $sub(S_0)$. This implies the termination of the algorithm.

Let us define the size of a term M, $size(M)$, as usual, and the size of a clause by $size(\mathsf{att}(M_1) \wedge \ldots \wedge \mathsf{att}(M_n) \rightarrow \mathsf{att}(M)) = size(M_1) + \ldots + size(M_n) + size(M)$. The hypotheses of clauses form a multiset, so when we compute $size(\sigma R)$ and the substitution σ maps several hypotheses to the same fact, this fact is counted several times in *size*. Intuitively, the size of clauses can increase during resolution, because the unification can instantiate terms. However, the size of their corresponding closed instance in $sub(S_0)$ decreases.

Proposition 2. *Assuming a weakly tagged program (Definition 5) and* $\mathcal{R}_0 = \mathcal{R}_{\mathrm{CryptoConstr}} \cup \mathcal{R}_{\mathrm{CryptoDestr}} \cup \mathcal{R}_{\mathrm{Prot}}$*, the computation of saturate*$(\mathcal{R}_0)$ *terminates.*

Proof. We show by induction that all rules R generated from $\mathcal{R}_0$ either are in $\mathcal{R}_{\mathrm{CryptoConstr}} \cup \mathcal{R}'_{\mathrm{CryptoDestr}}$, or are such that the terms of R are non-data, fully tagged, and mapped to $sub(S_0)$ by a substitution σ, i.e., $terms(\sigma R) \subseteq sub(S_0)$.

First, we can easily show that all rules in $condense(\mathcal{R}_0)$ satisfy this property.

If we combine by resolution two rules in $\mathcal{R}_{\mathrm{CryptoConstr}} \cup \mathcal{R}'_{\mathrm{CryptoDestr}}$, we in fact combine one rule of $\mathcal{R}_{\mathrm{CryptoConstr}}$ with one rule of $\mathcal{R}_{\mathrm{CryptoDestr}}$. The resulting rule is a tautology by condition W1, so it is eliminated immediately.

Otherwise, we combine by resolution a rule R such that the terms of R are non-data and fully tagged, and there exists a substitution σ such that $terms(\sigma R) \subseteq sub(S_0)$, with a rule R' such that one of 1., 2., or 3. holds.

1. The terms of R' are non-data and fully tagged, there exists a substitution σ' such that $terms(\sigma' R') \subseteq sub(S_0)$, and $sel(R') = \emptyset$ (in which case $sel(R) \neq \emptyset$).
2. $R' \in \mathcal{R}_{\text{CryptoConstr}}$.
3. $R' \in \mathcal{R}'_{\text{CryptoDestr}}$.

Let R'' be the rule obtained by resolution of R and R'. We show that the terms of R'' are fully tagged, and there exists a substitution σ'' such that $terms(\sigma'' R'') \subseteq sub(S_0)$ and $size(\sigma'' R'') < size(\sigma R)$.

Let $M_0, \ldots, M_n$ be the terms of R, $\mathsf{att}(M_0)$ being the atom of R on which we resolve. In all cases, the terms of R' are $M', x_1, \ldots, x_{n'}$, the variables $x_1, \ldots, x_{n'}$ occur in M' and are pairwise distinct variables, and $\mathsf{att}(M')$ is the atom of R' on which we resolve. (In case 1, because $sel(R') = \emptyset$ and by the optimizations *elimattx* and *elimdup*; in case 2, by definition of constructor rules; in case 3, by W1.) The terms M_0 and M' unify, let σ_u be their most general unifier. Then the terms of R'' are $\sigma_u x_1, \ldots, \sigma_u x_{n'}, \sigma_u M_1, \ldots, \sigma_u M_n$. By the choice of the selection function, the terms M_0 and M' are not variables.

We know that $\sigma M_0, \ldots, \sigma M_n$ are in $sub(S_0)$. We show that there exists σ' such that $\sigma M_0 = \sigma' M'$.

- In case 1, there exists σ' such that $\sigma' M' \in sub(S_0)$. The terms M_0 and M' are non-data fully tagged, so all their non-variable non-data subterms are top-tagged. In particular, since they are not variables, M_0 and M' themselves are top-tagged, i.e., M_0 is an instance of some $N_0 \in tagGen$ and M' is an instance of some $N'_0 \in tagGen$. Since M_0 and M' unify, so do N_0 and N'_0, $\sigma' M'$ is an instance of N'_0 in $sub(S_0)$, σM_0 is an instance of N_0 in $sub(S_0)$, so by condition W3, $\sigma' M' = \sigma M_0$.
- In case 2, M' is of the form $f(x_1, \ldots, x_{n'})$. Since M_0 is not a variable and unifies with M', M_0 has root symbol f, so σM_0 is an instance of M'.
- In case 3, by condition W1, M' is more general than every term in $sub(S_0)$ with the same root symbol, hence the instance σM_0 of the term M_0 that is unifiable with M' and thus has the same root symbol.

The substitution equal to σ on the variables of R and to σ' on the variables of R' is then a unifier of M_0 and M'. Since σ_u is the most general unifier, there exists σ'' such that $\sigma'' \sigma_u$ is equal to σ on the variables of R, and to σ' on the variables of R'. Thus the terms of $\sigma'' R''$ are $\sigma' x_1, \ldots, \sigma' x_{n'}, \sigma M_1, \ldots, \sigma M_n$. The terms $\sigma' x_1, \ldots, \sigma' x_{n'}$ are subterms of $\sigma' M' = \sigma M_0$ which is in $sub(S_0)$, so they are also in $sub(S_0)$. So all terms of $\sigma'' R''$ are in $sub(S_0)$.

Moreover, $size(\sigma'' R'') < size(\sigma' R')$. Indeed, $x_1, \ldots, x_{n'}$ occur in M' and are different variables. So $\sigma' x_1, \ldots, \sigma' x_{n'}$ are disjoint subterms of $\sigma' M'$, and M' does not consist of only a variable, so $size(\sigma' x_1) + \ldots + size(\sigma' x_{n'}) < size(\sigma' M') = size(\sigma M_0)$, and $size(\sigma'' R'') < size(\sigma M_0) + \ldots + size(\sigma M_n) = size(\sigma R)$.

We show that the terms of R'' are fully tagged.

- In case 1, since σ_u is the most general unifier of fully tagged terms, we can show that, for all x, $\sigma_u x$ is fully tagged, so for all fully tagged terms M, we can show that $\sigma_u M$ is fully tagged, so the terms of R'' are fully tagged.

- In case 2, for x among $x_1, \ldots, x_{n'}$, $\sigma_u x$ is a subterm of M_0, so is fully tagged. The terms $\sigma_u M_1, \ldots, \sigma_u M_n$ are equal to $M_1, \ldots, M_n$, also fully tagged.
- In case 3, $M' = f(M'_1, \ldots, M'_m)$ and $M_0 = f(M''_1, \ldots, M''_m)$, so σ_u is also the most general unifier of the pairs (M'_1, M''_1), $\ldots$, (M'_m, M''_m) of fully tagged terms. So we conclude as in case 1.

Finally, the terms of R'' are fully tagged, $terms(\sigma'' R'') \subseteq sub(S_0)$, and $size(\sigma'' R'') < size(\sigma R)$.

Then it is easy to show that all rules $R_s \in simplify(R'')$ obtained after simplification of R'' have non-data fully tagged terms and satisfy $terms(\sigma'' R_s) \subseteq sub(S_0)$, and $size(\sigma'' R_s) < size(\sigma R)$. Indeed, all rules in $decomp(R'')$ satisfy this property. (The decomposition of data constructors transforms fully tagged terms into non-data fully tagged terms.) This property is preserved by *elimdup* and *elimattx*.

Therefore, for all generated rules R, there exists σ such that $size(\sigma R)$ is smaller than the maximum initial value of $size(\sigma R)$ for a rule of the protocol. There is a finite number of such rules (since $size(R) \leq size(\sigma R)$). So the algorithm terminates. □

The termination of the backward depth-first search for closed facts is easy to show, for example by a proof similar to that of [4]. Essentially, the size of the goal decreases, because the size of the hypotheses of each clause is smaller than the size of the conclusion. (Recall that all terms of hypotheses of clauses of $saturate(\mathcal{R}_0) \cup \mathcal{R}_{\mathrm{DataConstr}}$ are variables that occur in the conclusion.) So we obtain:

Theorem 1. *The resolution-based verification algorithm terminates for weakly tagged programs and closed facts.*

As a corollary, by Proposition 1, we obtain the same result for tagged protocol programs, when $Params_{\mathsf{host}}$ and $Params_{\mathsf{pk}}$ have at most one element.

5.2 Handling Several Keys

The extension to several arguments of pk or of host requires an additional step. We define a homomorphism h from terms to terms that replaces all elements of $Params_{\mathsf{pk}}$ and of $Params_{\mathsf{host}}$ with a special constant k_0. We extend h to facts, clauses, and sets of clauses naturally. For the protocol program $h(\mathcal{R}_{\mathrm{Prot}})$, $Params_{\mathsf{pk}}$ and $Params_{\mathsf{host}}$ each have at most one element. So by Proposition 1, when $\mathcal{R}_{\mathrm{Prot}}$ is a tagged protocol program, $h(\mathcal{R}_{\mathrm{Prot}})$ is a weakly tagged program.

Let $\mathcal{R}_{\mathrm{Prot}}$ be any program such that $h(\mathcal{R}_{\mathrm{Prot}})$ is a weakly tagged program. We consider a "less optimized algorithm" in which elimination of duplicate hypotheses and of tautologies are performed only for facts of the form $\mathsf{att}(x)$ and elimination of subsumed clauses is performed only for the condensing of rules of $\mathcal{R}_{\mathrm{CryptoDestr}}$. We observe that Theorem 1 holds also for the less optimized algorithm, with the same proof, so this algorithm terminates on $h(\mathcal{R}_{\mathrm{Prot}})$. All resolution steps possible for the less optimized algorithm applied to $\mathcal{R}_{\mathrm{Prot}}$ are

possible for the less optimized algorithm applied to $h(\mathcal{R}_{\text{Prot}})$ as well (more terms are unifiable, and the remaining optimizations of the less optimized algorithm commute with the application of h). Then the less optimized algorithm terminates on $\mathcal{R}_{\text{Prot}}$. We can show that then the original, fully optimized algorithm also terminates.

In particular, the algorithm terminates for all tagged protocol programs and for implicitly tagged protocols, such as the Yahalom protocol without tags by Remark 1.

Theorem 2. *The resolution-based verification algorithm terminates for tagged protocol programs and closed facts.*

We recall that a tagged protocol program may be obtained by translating a protocol after tagging, and that the algorithm checks the non-derivability of the closed fact $\mathsf{att}(M)$, which shows the secrecy of the message M.

Although, for tagged protocols, the worst-case complexity of the algorithm is exponential (we did not detail this result by lack of space), it is quite efficient in practice. It remains to be seen whether there exists a smaller class containing most interesting examples, and for which the algorithm is polynomial.

6 Related Work

The verification problem of cryptographic protocols is undecidable [13], so one either restricts the problem, or approximates it.

Decision procedures have been published for restricted cases. In the case of a bounded number of sessions, for protocols using public-key cryptography with atomic keys and shared-key cryptography, protocol insecurity is NP-complete [23], and decisions procedures appear in [11,19,23]. When messages are bounded and no nonces are created, secrecy is DEXPTIME-complete [13]. Strong syntactic restrictions on protocols also yield decidability: [10] for an extension of ping-pong protocols, [3] with a bound on the number of parallel sessions, and restricted matching on incoming messages (in particular, this matching should be linear and independent of previous messages). Model-checking also provides a decision technique for a bounded number of sessions [18] (with additional conditions). It has been extended, with approximations, to an unbounded number of sessions using data independence techniques [6,7,22], for sequential runs, or when the agents are "factorisable". (Essentially, a single run of the agent has to be split into several runs, such that each run contains only one fresh value.)

On the other hand, some analyses terminate for all protocols, but at the cost of approximations. For instance, control-flow analysis [20] runs in cubic time, but does not preserve relations between components of messages, hence introduces an important approximation. Interestingly, the proof that control flow analysis runs in cubic time also relies on the study of a particular class of Horn clauses. Techniques using tree automata [15] and rank functions [17] also provide a terminating but approximate analysis. Moreover, the computation algorithm of rank functions assumes atomic keys.

It has been shown in [16] that tagging prevents type flaw attacks. It may be possible to infer from [16] that the depth of closed terms can be bounded in the search for an attack. This yields the decidability by exhaustive search, but does not imply the termination of our algorithm (in particular, because clauses can have an unbounded number of hypotheses, so there is an infinite number of clauses with a bounded term depth).

As for the approach based on Horn clauses, Weidenbach [24] already gave conditions under which his algorithm terminates. These conditions may give some idea of why the algorithm terminates on protocols. They do not seem to apply to many examples of cryptographic protocols.

Other techniques such as theorem proving [21] in general require human intervention, even if some cases can be proved automatically [9,12]. In general, typing [1,14] requires human intervention in the form of type annotations, that can be automatically checked. The idea of tagging already appears in [14] in a different context (tagged union types).

7 Conclusion

We have given the theory behind an experimental observation: tagging a protocol enforces the termination of the resolution-based verification technique used. Our work has an obvious consequence to protocol design, namely when one agrees that a design choice in view of a-posteriori verification is desirable.

Our termination result for *weakly* tagged protocols explains only in part another experimental observation, namely the termination for protocols without explicit tags. Although many of those are weakly tagged, some of them are not (for instance, the Needham-Schroeder public key protocol). The existence of a termination condition that applies also to those cases is open.

References

1. M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. In *29th ACM Symp. on Principles of Programming Languages (POPL'02)*, pages 33–44, Jan. 2002.
2. M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996.
3. R. Amadio and W. Charatonik. On Name Generation and Set-Based Analysis in the Dolev-Yao Model. In *CONCUR'02 - Concurrency Theory*, volume 2421 of *LNCS*, pages 499–514. Springer, Aug. 2002.
4. B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, June 2001.
5. B. Blanchet. From Secrecy to Authenticity in Security Protocols. In *9th Internat. Static Analysis Symposium (SAS'02)*, volume 2477 of *LNCS*, pages 342–359. Springer, Sept. 2002.
6. P. Broadfoot, G. Lowe, and B. Roscoe. Automating Data Independence. In *6th European Symp. on Research in Computer Security (ESORICS'00)*, volume 1895 of *LNCS*, pages 175–190. Springer, Oct. 2000.

7. P. J. Broadfoot and A. W. Roscoe. Capturing Parallel Attacks within the Data Independence Framework. In *15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 147–159, June 2002.
8. M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.
9. E. Cohen. TAPS: A First-Order Verifier for Cryptographic Protocols. In *13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 144–158, 2000.
10. H. Comon, V. Cortier, and J. Mitchell. Tree Automata with One Memory, Set Constraints, and Ping-Pong Protocols. In *Automata, Languages and Programming, 28th Internat. Colloq., ICALP'01*, volume 2076 of *LNCS*, pages 682–693. Springer, July 2001.
11. R. Corin and S. Etalle. An Improved Constraint-Based System for the Verification of Security Protocols. In *9th Internat. Static Analysis Symposium (SAS'02)*, volume 2477 of *LNCS*, pages 326–341. Springer, Sept. 2002.
12. V. Cortier, J. Millen, and H. Rueß. Proving secrecy is easy enough. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 97–108, June 2001.
13. N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols (FMSP'99)*, July 1999.
14. A. Gordon and A. Jeffrey. Authenticity by Typing for Security Protocols. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 145–159, June 2001.
15. J. Goubault-Larrecq. A Method for Automatic Cryptographic Protocol Verification (Extended Abstract), invited paper. In *5th Internat. Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'00)*, volume 1800 of *LNCS*, pages 977–984. Springer, May 2000.
16. J. Heather, G. Lowe, and S. Schneider. How to Prevent Type Flaw Attacks on Security Protocols. In *13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 255–268, July 2000.
17. J. Heather and S. Schneider. Towards automatic verification of authentication protocols on an unbounded network. In *13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 132–143, July 2000.
18. G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996.
19. J. Millen and V. Shmatikov. Constraint Solving for Bounded-Process Cryptographic Protocol Analysis. In *8th ACM Conf. on Computer and Communications Security (CCS'01)*, pages 166–175, 2001.
20. F. Nielson, H. R. Nielson, and H. Seidl. Cryptographic Analysis in Cubic Time. In *TOSCA 2001 - Theory of Concurrency, Higher Order Languages and Types*, volume 62 of *ENTCS*, Nov. 2001.
21. L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *J. Computer Security*, 6(1–2):85–128, 1998.
22. A. W. Roscoe and P. J. Broadfoot. Proving Security Protocols with Model Checkers by Data Independence Techniques. *J. Computer Security*, 7(2, 3):147–190, 1999.
23. M. Rusinovitch and M. Turuani. Protocol Insecurity with Finite Number of Sessions is NP-complete. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 174–187, June 2001.
24. C. Weidenbach. Towards an Automatic Analysis of Security Protocols in First-Order Logic. In *16th Internat. Conf. on Automated Deduction (CADE-16)*, volume 1632 of *LNAI*, pages 314–328. Springer, July 1999.

A Normalisation Result for Higher-Order Calculi with Explicit Substitutions

Eduardo Bonelli[1,2]

[1] LIFIA, Facultad de Informática, Universidad Nacional de La Plata, 50 y 115, La Plata (1900), Buenos Aires, Argentina.
[2] Department of Computer Science, Stevens Institute of Technology, Hoboken, NJ 07030, USA.
ebonelli@cs.stevens-tech.edu

Abstract. Explicit substitutions (ES) were introduced as a bridge between the theory of rewrite systems with binders and substitution, such as the λ-calculus, and their implementation. In a seminal paper P.-A. Melliès observed that the dynamical properties of a rewrite system and its ES-based implementation may not coincide: he showed that a strongly normalising term (i.e. one which does not admit infinite derivations) in the λ-calculus may lose this status in its ES-based implementation. This paper studies normalisation for the latter systems in the general setting of higher-order rewriting: Based on recent work extending the theory of needed strategies to non-orthogonal rewrite systems we show that needed strategies normalise in the ES-based implementation of any orthogonal pattern higher-order rewrite system.

1 Introduction

This paper studies normalisation for calculi of explicit substitutions (ES) implementing higher-order term rewrite systems (HORS). The latter are rewrite systems in which binders and substitution are present, the λ-calculus [Bar84] being a typical example. A recent approach to the implementation of HORS is the use of ES [ACCL91, BBLRD96, KR95, DG01]. ES were introduced as a bridge between HORS and their concrete implementations. Their close relation with abstract reduction machines [ACCL91, HMP96, BBLRD96] allows us to speak of ES-based *implementations* of HORS. The idea behind these implementations is that the complex notion of substitution is promoted to the object-level by introducing new operators into the language in order to compute substitutions explicitly. This allows HORS to be expressed as more fine-grained (first-order) rewrite systems in which no complex substitution nor binders are present. Such a process may be applied to any HORS [BKR01]. As an example Fig. 1 shows the rules of $\lambda\sigma$ [ACCL91], a calculus of ES implementing the λ-calculus (based on de Bruijn indices notation [dB72] in order to discard annoying issues related to the renaming of variables[1]).

[1] Variables in terms are represented as positive integers. Eg. $\lambda x.x$ is represented as $\lambda 1$, $\lambda x.\lambda y.x$ as $\lambda\lambda 2$, and $\lambda x.y$ as $\lambda 2$.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 153–168, 2003.

$$
\begin{array}{lll}
\text{Terms} & X := n \mid XX \mid \lambda X \mid X[s] & \\
\text{Substitutions} & s := id \mid \uparrow \mid X \cdot s \mid s \circ s &
\end{array}
$$

$$
\begin{array}{llllll}
(\lambda X)\, Y & \to_{Beta} & X[Y \cdot id] & & & \\
(X\, Y)[s] & \to_{App} & X[s]\, Y[s] & (X \cdot s) \circ t & \to_{Map} & X[t] \cdot (s \circ t) \\
(\lambda X)[s] & \to_{Lam} & \lambda X[1 \cdot (s \circ \uparrow)] & id \circ s & \to_{IdL} & s \\
X[s][t] & \to_{Clos} & X[s \circ t] & (s_1 \circ s_2) \circ s_3 & \to_{Ass} & s_1 \circ (s_2 \circ s_3) \\
1[X \cdot s] & \to_{VarCons} & X & \uparrow \circ (X \cdot s) & \to_{ShiftCons} & s \\
1[id] & \to_{VarId} & 1 & \uparrow \circ id & \to_{ShiftId} & \uparrow
\end{array}
$$

Fig. 1. The $\lambda\sigma$ calculus

An obstacle which arises when using ES for implementing HORS is that results on normalisation which hold in the higher-order rewriting setting may not be preserved in its implementation. A well-known example of this mismatch is due to Melliès [Mel95]: he exhibited a strongly normalising (typed) term in the λ-calculus for which the $\lambda\sigma$-calculus introduces an infinite derivation. However, the problem is not confined to the setting of λ-calculus but rather affects any HORS. For example the following well-typed Haskell program:

```
map (\x → (map id [ map id [true] ])) [ map id [true] ]
```

(where `id` abbreviates `\x → x`) is easily seen to be strongly normalising (and reduces to `[[[true]]]`), however its ES-based implementation (cf. Ex. 2) may introduce infinite derivations for it in a similar way to [Mel95] (see [Bon03]).

This mismatch calls for careful consideration of normalising strategies in the context of ES-based implementations of HORS. This paper studies normalisation in the latter systems based on *needed* strategies, a notion introduced in [HL91]. Needed strategies are those which rewrite redexes which are "needed" (cf. Section 2.2) in order to attain a normal form, assuming it exists. For eg. the underlined redex in $1[2 \cdot \underline{(\lambda 1)\, 1} \cdot id]$ is not "needed" in order to achieve a normal form since there is derivation, namely $1[2 \cdot (\lambda 1)\, 1 \cdot id] \to_{VarCons} 2$, that never reduces it. In fact the infinite $\lambda\sigma$-derivation of the aforementioned Haskell program takes place inside a substitution s in a term of the form $1[X \cdot s]$.

The literature on needed strategies for HORS has required the systems to be *orthogonal* [HL91, Mar92, GKK00][2]. A system is orthogonal if no conflicts (overlap) between redexes may arise. Neededness for orthogonal systems does not suffice in our setting since HORS (even orthogonal ones) are implemented as *non-orthogonal* systems in the ES-based approach. For eg., although the λ-calculus is orthogonal, $\lambda\sigma$ is not, as witnessed by the critical pair: $(\lambda X)[s]\, Y[s] \leftarrow_{App} ((\lambda X)\, Y)[s] \to_{Beta} X[Y \cdot id][s]$. However, recently an extension has been introduced for non-orthogonal systems [Mel96, Mel00]. Motivated by this work on needed derivations for non-orthogonal systems we prove the following new result: all needed strategies in ES-based implementations of arbitrary

[2] An exception is [Oos99] however only weakly orthogonal systems are studied.

orthogonal pattern HORS normalise. This extends the known result [Mel00] that needed strategies in $\lambda\sigma$ normalise.

As an example of (one of) the issues which must be revisited in the extended setting of HORS is a result [Mel00, Lem.6.4] which states that if the first redex in a *standard* $\lambda\sigma$-derivation occurs under a "λ" symbol, then the whole derivation does so too. A standard derivation is one in which computation takes place in an outside-in fashion (Def. 1). What makes "λ" so special in this regard in the $\lambda\sigma$-calculus is that creation of redexes above "λ", from below it, is not possible. We introduce the notion of *contributable symbol* in order to identify these special symbols in the ES-based implementation of arbitrary higher-order rewrite systems (under our definition "λ" is uncontributable), and prove an analogous result.

Structure of the paper. Section 2 reviews the ERS_{db} higher-order rewriting formalism and the theory of neededness for orthogonal systems together with Melliès' extension to non-orthogonal ones. Section 3 identifies the Standard-Projection Proposition as the only requirement for an orthogonal pattern HORS to verify normalisation of needed strategies. Section 4 is devoted to verifying that the latter holds for HORS. We then conclude and suggest possible future research directions.

2 Setting the Scene

2.1 The ERS_{db} Formalism and Its ES-based Implementations

The ERS_{db} formalism. ERS_{db} is a higher-order rewriting formalism based on de Bruijn indices notation [BKR00]. Rewrite rules are constructed from metaterms; metaterms are built from: de Bruijn *indices* $1, 2, \ldots$, *metavariables* $X_l, Y_l, Z_l, \ldots$ where l is a label (i.e. a finite sequence of symbols) over an alphabet of *binder indicators* $\alpha, \beta, \ldots$, *function symbols* $f, g, h, \ldots$ equipped with an arity n with $n \geq 0$, *binder symbols* $\lambda, \mu, \nu, \xi, \ldots$ equipped with a positive arity, and a *metasubstitution* operator $M[\![N]\!]$. *Terms* are metaterms without occurrences of metavariables nor metasubstitution. A *rewrite rule* is a pair of metaterms $L \to R$ such that: the head symbol of L is either a function or a binder symbol, all metavariables occurring in R also occur in L (disregarding labels), and there are no metasubstitutions in L. An ERS_{db} $\mathcal{R}$ is a set of rewrite rules. The $\lambda\sigma$-calculus of Fig. 1 is an ERS_{db} (as well as all first-order rewrite systems). Two other examples are:

Example 1 (ERS_{db} rewrite rules).

$app(\lambda X_\alpha, Y_\epsilon)$	$\to_{\beta_{db}}$	$X_\alpha[\![Y_\epsilon]\!]$
$map(\xi X_\alpha, nil)$	$\to_{map.1}$	nil
$map(\xi X_\alpha, cons(Y_\epsilon, Z_\epsilon))$	$\to_{map.2}$	$cons(X_\alpha[\![Y_\epsilon]\!], map(\xi X_\alpha, Z_\epsilon))$

A *rewrite step* is obtained by instantiating rewrite rules with valuations; the latter result from extending assignments (mappings from metavariables to terms) to the full set of metaterms and computing metasubstitutions $M[\![N]\!]$ by the

usual de Bruijn substitution [BKR00]. The labels in metavariables are used to restrict the set of valuations to "good" ones. For example, the classical η_{db} rule is written $\lambda(app(X_\alpha, 1)) \rightarrow_{\eta_{db}} X_\epsilon$. The X_α and X_ϵ indicate that a valuation is good if it assigns X_α and X_ϵ some term t in which 1-level indices do not occur free, for otherwise this index would be bound on the *LHS* and free on the *RHS*. Another example is $imply(\exists\forall X_{\alpha\beta}, \forall\exists X_{\beta\alpha}) \rightarrow_{Comm} true$ where a good valuation must verify that if t is assigned to $X_{\alpha\beta}$ then the term resulting from t by interchanging 1 with 2-level indices must be assigned to $X_{\beta\alpha}$ [BKR00].

A *redex* r is a triple consisting of a term M, a valuation, and a position[3] p in M such that M at position p is an instance of the *LHS* of a rewrite rule via the valuation; the induced rewrite step is written $M \rightarrow_r N$. Letters $r, s, t, \ldots$ stand for redexes. We use $\rightarrow_{\mathcal{R}}$ for the rewrite step relation induced by an ERS_{db} $\mathcal{R}$ and $\twoheadrightarrow_{\mathcal{R}}$ for its reflexive-transitive closure. A *derivation* is a sequence of rewrite steps; we use $|\phi|$ for the length (number of rewrite steps) of a derivation ϕ; letters $\phi, \varphi, \ldots$ stand for derivations. If $r_1, \ldots, r_n$ are composable rewrite steps then $r_1; \ldots; r_n$ is the derivation resulting from composing them. Derivations that start (resp. end) at the same term are called coinitial (resp. cofinal).

ES-based implementations of HORS. Any ERS_{db} may be implemented as a first-order rewrite system with the use of ES [BKR01, Bon01]. The implementation process (cf. Rem. 1) goes about dropping labels in metavariables and replacing metasubstitution operators $\bullet[\![\bullet]\!]$ in rewrite rules with explicit substitutions $\bullet[\bullet \cdot id]$. Also, new rules - the substitution calculus - are added in order to define the behavior of the new explicit substitutions; roughly, this calculus is in charge of propagating substitutions until they reach indices and then discarding the substitutions or replacing the indices. In this paper we use the σ-calculus [ACCL91] as substitution calculus[4], its rules have been presented in Fig. 1 (disregarding *Beta*); it is confluent [ACCL91] and strongly normalising [CHR92]. If $\mathcal{R}$ is an ERS_{db} then we write $\mathcal{R}^{\mathsf{ES}}_{\sigma}$ for its ES-based implementation and refer to it as an *implementation* (of $\mathcal{R}$).

Example 2.

$(\beta_{db})^{\mathsf{ES}}_{\sigma}$=$\lambda\sigma$	
$map^{\mathsf{ES}}_{\sigma} = map.1^{\mathsf{ES}} \cup map.2^{\mathsf{ES}} \cup \sigma$ where:	
$map(\xi X, nil)$	$\rightarrow_{ES(map.1)}\ nil$
$map(\xi X, cons(Y, Z))$	$\rightarrow_{ES(map.2)}\ cons(X[Y \cdot id], map(\xi X, Z))$

Two basic properties of ES-based implementations of HORS are *Simulation* (if $M \rightarrow_{\mathcal{R}} N$ then for some M', $M \rightarrow_{\mathcal{R}^{\mathsf{ES}}} M' \twoheadrightarrow_{\sigma} \sigma(N)$, where $\sigma(N)$ denotes the σ-normal form of N) and *Projection* ($M \rightarrow_{\mathcal{R}^{\mathsf{ES}}_{\sigma}} N$ then $\sigma(M) \twoheadrightarrow_{\mathcal{R}} \sigma(N)$). For eg. $map(\xi(cons(1, nil)), cons(2, nil)) \rightarrow_{map.2} cons(cons(2, nil), map(\xi(cons(1, nil)), nil))$ may be simulated in its ES-based implementation as:

[3] As usual, positions are paths in (terms represented as) trees [DJ90, BN98].

[4] The rules *App* and *Lam* in Fig. 1 are present because $\lambda\sigma$ implements β_{db}; in the general case we would have $f(X_1, \ldots, X_n)[s] \rightarrow_{Funcf} f(X_1[s], \ldots, X_n[s])$ for each function symbol f and $\xi(X_1, \ldots, X_n)[s] \rightarrow_{Bind_\xi} \xi(X_1[1 \cdot (s \circ \uparrow)], \ldots, X_n[1 \cdot (s \circ \uparrow)])$ for each binder symbol ξ.

$$
\begin{array}{ll}
 & map(\xi(cons(1, nil)), cons(2, nil)) \\
\rightarrow_{ES(map.2)} & cons(cons(1, nil)[2 \cdot id], map(\xi(cons(1, nil)), nil)) \\
\rightarrow_{Func_{cons}} & cons(cons(1[2 \cdot id], nil[2 \cdot id]), map(\xi(cons(1, nil)), nil)) \\
\rightarrow_{VarCons} & cons(cons(2, nil[2 \cdot id]), map(\xi(cons(1, nil)), nil)) \\
\rightarrow_{Func_{nil}} & cons(cons(2, nil), map(\xi(cons(1, nil)), nil))
\end{array}
$$

Remark 1. We restrict attention to a subset of ERS_{db}, namely the class of *Pattern* ERS_{db}($PERS_{db}$). This corresponds to the usual requirement that *LHS* of rules be higher-order patterns in other rewrite formalisms [Mil91, Nip91, KOvR93]. $\mathcal{R}$ is defined to be a pattern ERS_{db} if its ES-based implementation does not contain explicit substitutions on the *LHS*. For example, η_{db} is translated to $\lambda(app(X[\uparrow], 1)) \rightarrow_{ES(\eta_{db})} X$. And *Comm* to $imply(\exists\forall X, \forall\exists X[2 \cdot 1 \cdot (\uparrow \circ \uparrow)]) \rightarrow true$ [BKR01]. Thus η_{db} and *Comm* are not pattern ERS_{db}; those of Ex. 1 are. The former exhibit the fact that when translating to an ES-based setting, higher-order matching may not always be coded as syntactic matching. The "occurs check" imposed by η_{db} or the "commutation of indices check" imposed by *Comm* are complex features of higher-order matching that require further machinery (matching *modulo* the calculus of ES) in order to be mimicked in a first-order setting. This is why we consider pattern ERS_{db}.

2.2 Standardisation and Neededness

The notion of standard derivations in rewriting[5] may be formalised using the *redex-permutation* approach [Klo80, Mel96]. We revisit this approach *very* briefly.

Given two non-overlapping redexes r, s in some term M we define the notion of a redex *residual* of r after contracting s (written r/s) with an example. In $M = \underline{(\lambda\lambda(2\,2))\,((\lambda 1)\,2)}\,3 \rightarrow_{\beta_{db}} (\lambda((\lambda 1)\,3)\,((\lambda 1)\,3))\,3 = N$, the residuals of $r = ((\lambda 1)\,2)$ in M after contracting the underlined redex s are the two copies of $(\lambda 1)\,3$ in N. Note that s has no residuals in N (i.e. for any s, $s/s = \emptyset$), and also that r/s is a finite set of redexes. The outermost redex in N is said to be *created* since it is not the residual of any redex in M. If U_M is a finite set of non-overlapping redexes ($r, s \in U_M$ implies r does not overlap s) in M and s is a redex in M, then $U_M/s = \{v | \exists u \in U_M, v \in u/s\}$. The residuals of r after a derivation $r_1; \ldots; r_n$ coinitial with it is defined as $((r/r_1)/r_2)\ldots/r_n$. A *development* of U_M is a derivation $\phi = r_1; \ldots; r_n$ s.t. $r_i \in U_M/(r_1; \ldots; r_{i-1})$ for $i \in 1..n$ and $U_M/\phi = \emptyset$ (if this last condition is not satisfied we say ϕ is a *partial development* of U_M). A well-known result called Finite Developments states that all partial developments are finite [Bar84, Oos94]. This allows one to prove the following:

Proposition 1 (Parallel Moves or Basic Tile Lemma [Bar84, HL91]). *Given two non-overlapping redexes r, s in some term M, the divergence resulting from contracting r and s may be settled by developing their corresponding residuals (Fig. 2(a)). Moreover, for any u in M, $u/(r; s/r) = u/(s; r/s)$.*

[5] In the sequel we restrict attention to left-linear rewrite systems: variables occur at most once in *LHS*s.

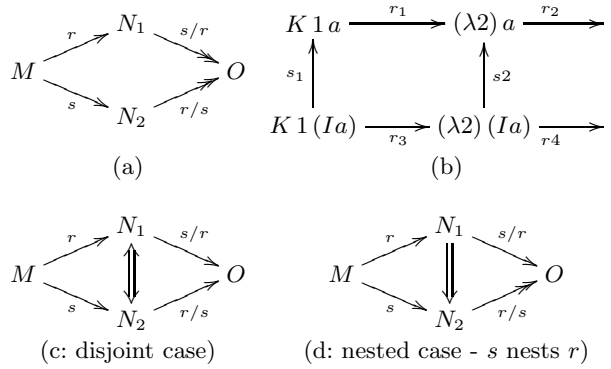


Fig. 2. Tiles

Fig. 2(b) shows two basic tiles in the λ-calculus, where $K = \lambda\lambda 2$. As depicted, these basic tiles may be "glued" in order to construct tilings between some coinitial and cofinal derivations ϕ and φ, in which case we write $\phi \equiv \varphi$ and say that ϕ and φ are Lévy-Permutation Equivalent [Lév78, Klo80, Mel96]. For example, $s_1; r_1; r_2 \equiv r_3; s_2; r_2$, however $I(\underline{I\,1}) \rightarrow_{\beta_{db}} I\,1 \not\equiv \underline{I(I\,1)} \rightarrow_{\beta_{db}} I\,1$, where $I = \lambda 1$. Moreover, by comparing the relative positions of r and s ((1) disjoint, left tile in Fig. 2(b) and (2) nested - a redex s nests r when the position of s is a prefix of the position of r - right tile in Fig. 2(b)) we can orient these tiles as indicated in Fig. 2(c,d) and define standard derivations.

Definition 1 (Standard derivation).

1. *Let r, s be cointial redexes. If they are disjoint, then $r; s/r \diamond s; r/s$ models a reversible tile; if s nests r, then $r; s/r \rhd s; r/s$ models an irreversible one. A reversible step ($\overset{\mathtt{r}}{\Rightarrow}$) is defined as $\phi_1; r; s/r; \phi_2 \overset{\mathtt{r}}{\Rightarrow} \phi_1; s; r/s; \phi_2$ where $r; s/r \diamond s; r/s$; an irreversible one ($\overset{\mathtt{i}}{\Rightarrow}$) as $\phi_1; r; s/r; \phi_2 \overset{\mathtt{i}}{\Rightarrow} \phi_1; s; r/s; \phi_2$ where $r; s/r \rhd s; r/s$. $\Rightarrow$ denotes the least reflexive-transitive closure of $\overset{\mathtt{r}}{\Rightarrow} \cup \overset{\mathtt{i}}{\Rightarrow}$ and $\simeq$ is the least equivalence relation containing $\overset{\mathtt{r}}{\Rightarrow}$.*
2. *Two coinitial and cofinal derivations ϕ, φ are Lévy-permutation equivalent, if $\phi \equiv \varphi$ where $\equiv$ is the least equivalence relation containing $\Rightarrow$.*
3. *A derivation ϕ is standard if it is minimal in the following sense: there is no sequence of the form $\phi = \phi_0 \overset{\mathtt{r}}{\Rightarrow} \ldots \overset{\mathtt{r}}{\Rightarrow} \phi_{k-1} \overset{\mathtt{i}}{\Rightarrow} \phi_k$, where $k \geq 1$.*

For example $r_3; r_4$ is standard, but $r_3; s_2; r_2$ is not. The standardisation theorem states that every derivation may be standardised into a unique standard derivation by oriented tiling:

Theorem 1 (Standardisation [Lév78, Bar84, HL91, Mel96]).

1. *(Existence) For any ϕ there exists a standard derivation φ s.t. $\phi \Rightarrow \varphi$.*
2. *(Unicity) If $\varphi_1 \equiv \phi$ and $\varphi_2 \equiv \phi$ and $\varphi_{1,2}$ are standard, then $\varphi_1 \simeq \varphi_2$.*

Let $\mathtt{std}(\phi)$ stand for the (unique modulo $\simeq$) standard derivation in the $\equiv$-equivalence class of ϕ. For example, $\mathtt{std}(s_1; r_1; r_2) = \mathtt{std}(r_3; s_2; r_2) = r_3; r_4$ in Fig. 2(b).

Standard derivations are used to show that needed strategies are normalising in *orthogonal systems*. A rewrite system is orthogonal if any pair of coinitial redexes r, s do not overlap. Define r in M to be a *needed redex* if it has at least one residual in any coinitial derivation ϕ, unless ϕ contracts a residual of r [Mar92]. For example, for the rewriting system $\{f(X_\epsilon, b) \to c,\ a \to b\}$ the right occurrence of a in $f(a, a)$ is needed but not the left one. A needed rewrite strategy is one that only selects needed redexes. By defining a measure $|M|$ as "the length of the unique standard derivation (modulo $\simeq$) to M's normal form" it may be shown [HL91, Mel96] that if $M \to_r N$ for some needed redex r, then $|M| > |N|$; hence needed strategies normalise in orthogonal rewrite systems. This measure is well-defined since in orthogonal systems any two coinitial derivations to (the unique) normal form may be tiled [HL91, Mel96].

In the case of non-orthogonal systems the notion of needed redex requires revision. Indeed, in $\mathcal{R} = \{a \to_r a,\ a \to_s b\}$ the derivation to normal form $\phi : a \to_s b$ leaves no residual of r. However one cannot conclude that r is not needed since although r is not reduced in ϕ a redex which overlaps with r has. Thus the notion of needed redex is extended to needed derivations as follows:

Definition 2 (Needed derivations in non-orthogonal systems [Mel00]). *$\phi : M \twoheadrightarrow N$ is needed in a non-orthogonal rewrite system if $|\mathtt{std}(\phi; \psi)| > |\mathtt{std}(\psi)|$ for any term P and any derivation $\psi : N \twoheadrightarrow P$.*

Note that now $a \to_r a$ is needed in the aforementioned example. The concept of needed redexes is extended to that of derivations since, in contrast to orthogonal systems, terms in non-orthogonal ones may not have needed redexes. For example, in $\{xor(true, X_\epsilon) \to_L true, xor(X_\epsilon, true) \to_R true, \Omega \to_\Omega true\}$ the term $xor(\Omega, \Omega)$ has no needed redexes [Klo92].

Needed derivations get us "closer" to a normal form, however the aforementioned measure for orthogonal systems is no longer well-defined: there may be two or more $\equiv$-distinct normalising derivations. Eg. $\phi_1 : a \to_r a \to_s b$, $\phi_2 : a \to_r a \to_r a \to_s b$, $\phi_3 : a \to_r a \to_r a \to_r a \to_s b$, ..., etc. are $\equiv$-distinct normalising derivations since each r-step creates a new copy of a. In [Mel00] such badly-behaved systems are discarded by requiring the following property to be fulfilled: A *normalisation cone*[6] for a term M is a family $\{\psi_i : M \twoheadrightarrow N \mid i \in I_M\}$ of normalising derivations such that every normalising derivation $\phi : M \twoheadrightarrow N$ is Lévy-permutation equivalent to a unique derivation ψ_i (i.e. $\exists! i \in I_M$ s.t. $\phi \equiv \psi_i$). A rewrite system enjoys *finite normalisation cones* (*FNC*) when there exists a *finite* normalisation cone for any term M.

Redefining the measure of a term $|M|$ to be "the length of the longest standard derivation in M's cone to M's normal form" allows one to show [Mel00] that

[6] This definition differs slightly from [Mel00, Def.4.8] since we make use of the fact that ES-based implementations of orthogonal HORS are confluent [BKR01].

if $\phi : M \twoheadrightarrow N$ is a needed derivation, then $|M| > |N|$; hence needed strategies normalise in non-orthogonal rewrite systems satisfying *FNC*.

3 *FNC* for ES-based Implementations of HORS

It is therefore of interest to identify conditions guaranteeing that ES-based implementations of HORS enjoy finite normalisation cones. In [Mel00] the *FNC* property is shown for $\lambda\sigma$; this result relies on two conditions: (1) if ϕ is a standard derivation in $\lambda\sigma$ ending in a σ-normal form (cf. Sect. 4), then $\sigma(\phi)$ is standard in the λ-calculus, and (2) needed strategies are normalising for the λ-calculus; $\sigma(\phi)$ is obtained by mapping each $\lambda\sigma$-rewrite step in ϕ to its "corresponding" or "projected", if any, rewrite step in λ (see Stage 2, Sect. 4). Eg. if $\phi : (1\,1)[(\lambda 1)\,2 \cdot id] \to_{Beta} (1\,1)[1[2 \cdot id] \cdot id]$ then $\sigma(\phi)$ takes the form:

$$((\lambda 1)\,2)\,((\lambda 1)\,2) \to_{\beta_{db}} 2\,((\lambda 1)\,2) \to_{\beta_{db}} 2\,2$$

In this paper we show that the *FNC* property holds for the ES-based implementation of *arbitrary* orthogonal $PERS_{db}$. This generalizes [Mel00, Thm.7.1], proved for the λ-calculus. The proof follows the same lines as in [Mel00]; it relies on our meeting requirement (1), namely

Proposition 2 (Std-Projection Proposition). *Let $\mathcal{R}$ be a left-linear $PERS_{db}$. Every standard derivation $\phi : M \twoheadrightarrow N$ in $\mathcal{R}_\sigma^{\mathsf{ES}}$ with N in σ-normal form is projected onto a standard derivation $\sigma(\phi) : \sigma(M) \twoheadrightarrow N$ in $\mathcal{R}$.*

Here is how *FNC* follows from the Std-Projection Proposition:

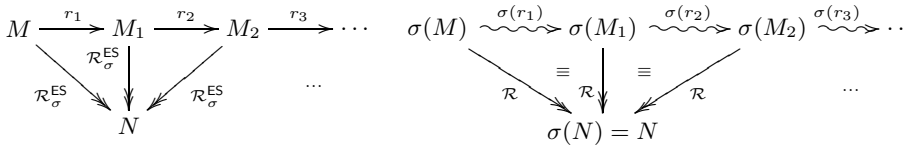
Proposition 3. *The ES-based implementation of any orthogonal $PERS_{db}$ $\mathcal{R}$ verifying the Std-Projection Proposition enjoys FNC: every closed $\mathcal{R}_\sigma^{\mathsf{ES}}$-term has FNC.*

Proof. Suppose, on the contrary, that there exists a closed $\mathcal{R}_\sigma^{\mathsf{ES}}$-term with an infinite number of normalising $\mathcal{R}_\sigma^{\mathsf{ES}}$-derivations, modulo Lévy-permutation equivalence. We may construct an infinite tree whose nodes are the standard derivations $M \twoheadrightarrow N$ which may be extended to normalising derivations $M \twoheadrightarrow N \twoheadrightarrow P$ where nodes are ordered by the prefix ordering, and by König's Lemma (since every $\mathcal{R}_\sigma^{\mathsf{ES}}$ term contains finite redexes) deduce the existence of an infinite derivation ϕ_∞. Moreover, since σ is strongly normalising we know that ϕ_∞ has an infinite number of $\mathcal{R}^{\mathsf{ES}}$-steps.

Let ϕ_∞ be of the form $M \to M_1 \to M_2 \to \ldots$. Every finite prefix $\phi_i : M \twoheadrightarrow M_i$ of ϕ_∞ may be extended to a standard normalising path $\chi_i : M \twoheadrightarrow M_i \twoheadrightarrow N$ (see below, left). And, by Prop. 2, each $\sigma(\chi_i) : \sigma(M) \twoheadrightarrow \sigma(M_i) \twoheadrightarrow \sigma(N) = N$ is a standard and normalising $\mathcal{R}$ derivation.

Since $\mathcal{R}$ is orthogonal all the normalising derivations $\sigma(\chi_i) : \sigma(M) \twoheadrightarrow \sigma(N) = N$ must be Lévy-permutation equivalent. Thus we have: $\sigma(\chi_1) \equiv \sigma(\chi_2) \equiv \sigma(\chi_3) \equiv \sigma(\chi_4) \equiv \ldots$. And from Thm. 1(2) and the fact that $\phi \simeq \varphi$ implies $|\phi| = |\varphi|$ we deduce that: $|\sigma(\chi_1)| = |\sigma(\chi_2)| = |\sigma(\chi_3)| = |\sigma(\chi_4)| = \ldots$

We reach a contradiction from the fact that there are an infinite number of $\mathcal{R}^{\mathsf{ES}}$ redexes in ϕ_∞ and that Prop. 2 projects $\mathcal{R}^{\mathsf{ES}}$ redexes to unique $\mathcal{R}$-redexes (see Stage 2, Sect. 4): For every $i > 0$ there is a $j > i$ such that $|\sigma(\chi_j)| > |\sigma(\chi_i)|$. See below, right, where the squiggly arrow $\sigma(r_i)$ is the identity if r_i is a σ redex and is an $\mathcal{R}$ redex if r_i is an $\mathcal{R}^{\mathsf{ES}}$ redex.



4 The Std-Projection Proposition

We now concentrate on the proof of the Std-Projection Proposition which proceeds by contradiction and is developed in three stages. Before continuing however, we remark that it is non-trivial. In fact, in the general case in which N is not required to be a σ-normal form it fails:

$$\chi : ((\lambda(11))1)[\underline{(\lambda 1)c} \cdot id] \to_{Beta} \underline{((\lambda(11))1)}[1[c \cdot id] \cdot id] \to_{Beta} (11)[1 \cdot id][1[c \cdot id] \cdot id]$$

χ is a standard $\lambda\sigma$-derivation, however $\sigma(\chi) : (\lambda(11))\underline{((\lambda 1)c)} \to_\beta \underline{(\lambda(11))c} \to_\beta cc$ is not standard in the λ-calculus.

Let χ be any standard $\mathcal{R}^{\mathsf{ES}}_\sigma$ derivation. The idea of the proof, inspired from [Mel00], is to show that every reversible (resp. irreversible) step in the projection $\sigma(\chi)$ of χ may be mimicked by 1 or more reversible (resp. reversible steps followed by 1 or more irreversible) steps in χ, the ES-based derivation. Hence we may conclude by reasoning by contradiction. Stage 1 of the proof shows why the projection of an $\mathcal{R}^{\mathsf{ES}}$ step results in a unique $\mathcal{R}$ step if χ ends in a σ-normal form, Stage 2 uses this fact to prove that reversible steps may be mimicked as explained above and Stage 3 considers irreversible steps.

Stage 1 (Substitution Zones)

First of all, note that χ consists of two kinds of rewrite steps: $\mathcal{R}^{\mathsf{ES}}$ steps and σ steps. We argue that it is not possible for a $\mathcal{R}^{\mathsf{ES}}$ step to take place inside a substitution if χ ends in a σ-normal form. The reason is that in that case the $\mathcal{R}^{\mathsf{ES}}$-redex would occur inside some term P in $P \cdot s$ and hence under the "$\cdot$" symbol. Since χ is standard, redexes reduced below a "$\cdot$" symbol cannot create redexes above it, and since N is a pure term we arrive at a contradiction. We formalise this argument below (Lemma 2).

Definition 3. *Given an implementation $\mathcal{R}^{\mathsf{ES}}_\sigma$ with Γ the set of function and binder symbols, we define $g \in \Gamma$ of arity n as uncontributable in $\mathcal{R}^{\mathsf{ES}}_\sigma$ if*

1. *either, g does not occur on the LHS of any rule in $\mathcal{R}^{\mathsf{ES}}_\sigma$,*
2. *or, g occurs on the LHS of a rule in $\mathcal{R}^{\mathsf{ES}}_\sigma$ only under the form $g(X_1, .., X_n)$ (i.e. it occurs applied to metavariables).*

A symbol in Γ which is not uncontributable is called contributable.

Example 3. The λ-symbol is an example of an uncontributable symbol in the $\lambda\sigma$-calculus, i.e. in $(\beta_{db})^{\mathsf{ES}}_{\sigma}$. Whereas, the application symbol is contributable in $\lambda\sigma$ due to the *Beta*-rule. Also, for any $PERS_{db}$ $\mathcal{R}$ the cons-symbol "$\cdot$" is uncontributable in $\mathcal{R}^{\mathsf{ES}}_{\sigma}$ since it is only the rules of σ that govern the behaviour of "$\cdot$".

The notion of uncontributable symbol attempts to capture those symbols from which reduction below it cannot create/erase redexes above it. Note that, although similar, this concept does not coincide with that of constructor symbol in a constructor TRS [Klo92]: given a constructor TRS there may be constructor symbols which are contributable (e.g. "s" in $f(s(s(x))) \to x$) and likewise there may be uncontributable symbols that are not constructor symbols (e.g. "f" in $f(x) \to a$).

We say that a derivation $r_1; \ldots; r_n$ *preserves* a position p when none of the redexes r_i is above p.

Lemma 1. *Let $\mathcal{R}^{\mathsf{ES}}_{\sigma}$ implement $\mathcal{R}$. Suppose that a position p is strictly above a redex $P \to_r Q$. Every standard derivation $\phi = r; \psi$ preserves p when:*

1. *either, p is a g-node for g an uncontributable symbol in $\mathcal{R}^{\mathsf{ES}}_{\sigma}$,*
2. *or, p is a g-node for g a function or binder symbol in $\mathcal{R}^{\mathsf{ES}}_{\sigma}$ and ψ is a σ-derivation.*

Proof. Given the standard derivation $\phi = r; \psi$ and the position p strictly above r two cases may arise: either ϕ preserves p (in which case we are done) or otherwise ϕ may be reorganized modulo $\simeq$ into a derivation $\phi_1; u; v; \phi_2$ such that ϕ_1 preserves the position p, the position p is strictly above a redex u, and a redex v is above p. We shall see that the latter case results in a contradiction. Note that the derivation $u; v$ cannot be standard, unless u creates v. Now, in at least the following two cases creation is not possible:

1. When p is the position of an uncontributable symbol in $\mathcal{R}^{\mathsf{ES}}_{\sigma}$. This follows from the fact that contraction of a redex below an uncontributable symbol may not create a redex above it.
2. When the position p is a function or binder symbol node and u is a σ-redex then an $\mathcal{R}^{\mathsf{ES}}$-redex must have been created, in other words, the only possible pattern of creation is when u is a $Func_f$-redex for some function symbol f or a $Bind_{\xi}$-redex for some binder symbol ξ and v is an $\mathcal{R}^{\mathsf{ES}}$-redex. For example, the pair $M = g(h(c)[id]) \to_{Func_h} g(h(c[id])) \to c$ where $\mathcal{R} = \{g(h(X)) \to c\}$, $p = \epsilon$ in M, the position at which v occurs in N is ϵ and the position at which u occurs in M is 1. Note that it is not possible for u to be an $\mathcal{R}^{\mathsf{ES}}_{\sigma}$-redex and v a σ-redex since σ-redexes above function or binder symbols cannot be created from below them.

The following key lemma states that the left argument of a "$\cdot$" symbol determines an "enclave" in a standard $\mathcal{R}^{\mathsf{ES}}_{\sigma}$-derivation to σ-normal form.

Lemma 2. *Let $\mathcal{R}_\sigma^{\mathsf{ES}}$ implement $\mathcal{R}$ and let $\phi : M \twoheadrightarrow N$ be a standard $\mathcal{R}_\sigma^{\mathsf{ES}}$-derivation with N in σ-normal form. Then no $\mathcal{R}_\sigma^{\mathsf{ES}}$-redex ever appears in the left argument of a substitution $P \cdot s$.*

Proof. By contradiction. Suppose there exists an r_i contracted in $\phi = r_1; \ldots; r_n$ inside the left argument P of a substitution $P \cdot s$. Since the "$\cdot$" symbol is uncontributable, then by Lemma 1(1) the derivation $r_i; \ldots; r_n$ preserves p. Since N is a pure term (i.e. has no explicit substitutions) we arrive at a contradiction.

Stage 2 (Reversible Steps)

So now we know that every $\mathcal{R}^{\mathsf{ES}}$ redex contracted in χ does not occur inside a substitution and hence that it has a unique *correspondent* $\mathcal{R}$-redex in $\sigma(\chi)$. This means that if $\sigma(\chi) = R_1; \ldots; R_o$, then there is a function $\rho : \{1, \ldots, o\} \to \{1, \ldots, n\}$ which associates to any $\mathcal{R}$-redex R_k in $\sigma(\chi)$ the unique $\mathcal{R}^{\mathsf{ES}}$-redex $r_{\rho(k)}$ in $\chi = r_1; \ldots; r_n$ to which it corresponds.

It should be mentioned that there are a number of subtle issues regarding the notion of "correspondence" that must be considered. Due to lack of space these issues have been relegated to the manuscript [Bon03], however we comment on them briefly.

- First of all, observe that it does not coincide with that of the residual relation since $\mathcal{R}^{\mathsf{ES}}$-redexes may be lost when traversed by substitutions: For example, the *Beta*-redex in M is lost in the following σ-derivation: $M = ((\lambda P)\, Q)[id] \to_{App} (\lambda P)[id]\, Q[id] \to_{Lam} (\lambda(P[1 \cdot id \circ \uparrow]))\, Q[id]$. Therefore, an appropriate notion of correspondent for tracing $\mathcal{R}^{\mathsf{ES}}$ redexes through σ derivations must be defined.
- Secondly, since σ rewriting may duplicate terms it must be proved that the correspondents of redexes not occurring inside substitutions are indeed unique.
- Finally, the following parametricity property for σ showing the absence of "syntactical coincidences" [HL91] must be verified: if $r_{\rho(i)}$ is a redex in M_i whose correspondent is R_i in $\sigma(M_i)$ then the definition of correspondent should not depend on any particular σ-derivation taking M_i to $\sigma(M_i)$.

Let R_k and R_{k+1} be two consecutive $\mathcal{R}$-redexes in $\sigma(\chi)$. Note that the $\mathcal{R}_\sigma^{\mathsf{ES}}$-derivation $r_i; \ldots; r_j = r_{\rho(k)+1}; \ldots; r_{\rho(k+1)-1}$ between $r_{\rho(k)}$ and $r_{\rho(k+1)}$ contracts only σ-redexes, as depicted below:

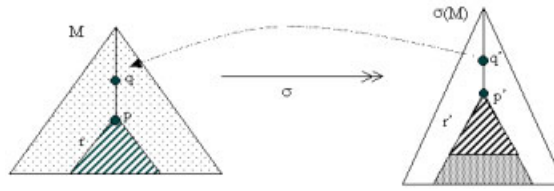
$$\begin{array}{ccccccc}
M_{i-1} & \xrightarrow[r_{\rho(k)}]{} & M_i & \overset{\sigma}{\underset{r_i;\ldots;r_j}{\twoheadrightarrow}} & M_j & \xrightarrow[r_{\rho(k+1)}]{} & M_{j+1} \\
\sigma\downarrow & & \sigma\downarrow & & \sigma\downarrow & & \sigma\downarrow \\
\sigma(M_{i-1}) & \xrightarrow[R_k]{} & \sigma(M_i) & = & \sigma(M_j) & \xrightarrow[R_{k+1}]{} & \sigma(M_{j+1})
\end{array}$$

We now show that every reversible standardisation step $\sigma(\chi) \overset{\mathtt{r}}{\Rightarrow} \omega$ in $\mathcal{R}$ may be mirrored as a non-empty series of reversible standardisation steps $\chi \overset{\mathtt{r}}{\Rightarrow} \ldots \overset{\mathtt{r}}{\Rightarrow} \phi$

in $\mathcal{R}_\sigma^{\mathsf{ES}}$, where $\sigma(\phi) = \omega$. It suffices to show that if R_k and R_{k+1} can be permuted using a reversible tile $(R_k; R_{k+1} \diamond R'_k; R'_{k+1})$, then a number of reversible steps may be applied to $r_{\rho(k)}; r_i; \ldots; r_j; r_{\rho(k+1)}$ yielding ϕ s.t $\sigma(\phi) = R'_k; R'_{k+1}$. However, first we need the notion of descendent of a position.

In the same way as the notion of residuals allow us to keep track of redexes along derivations, the notion of *descendent* allows us to keep track of positions (hence symbols) along derivations. Here is an example: if $\mathcal{R} = \{f(X_\epsilon, b) \to c\}$ and consider the rewrite step $M = f(f(a,b),c) \to f(c,c) = N$. The "$f$" at the root in M descends to the "f" at the root in N and the inner "f" in M (and the "b") descends to the "c" in N. The "a" in M, however, has no descendents in N. The inverse of the descendent relation is called the ancestor relation.

Remark 2 (Origins of positions outside substitutions). Let r be an $\mathcal{R}^{\mathsf{ES}}$-redex in M occurring at a position p not inside a substitution. As already noted, it has a unique corresponding $\mathcal{R}$-redex R in $\sigma(M)$ occurring at some position p'. Moreover, the following property on the ancestors of positions not inside substitutions holds: for every position q' in $\sigma(M)$ with $q' < p'$ (where $<$ is the prefix order), we have $q < p$ where q is a position in M of the same symbol and is the (unique) ancestor of q' as illustrated below. The fact that q' is unique follows from the observation that σ may not create new function or binder symbols.



Let us now turn to the proof of this stage. Suppose two $\mathcal{R}$-redexes R_k and R_{k+1} can be permuted using a reversible tile, that is, $R_k; R_{k+1} \diamond R'_k; R'_{k+1}$. We construct an $\mathcal{R}_\sigma^{\mathsf{ES}}$-derivation ϕ s.t. $\chi \simeq \phi$ and $\sigma(\phi) = R_1; \ldots; R'_k; R'_{k+1}; \ldots; R_p$. By Lemma 1(2), the derivation $r_i; \ldots; r_j$ preserves the position of any function or binder symbol strictly above $r_{\rho(k)}$. And, in particular, the lowest function or binder symbol g appearing above $R_k : \sigma(P) \to \sigma(Q)$ and R_{k+1} in the term $\sigma(P)$ which, by Remark 2, is strictly above $r_{\rho(k)}$ in P. Then the derivation $\psi = r_{\rho(k)}; r_i; \ldots; r_j; r_{\rho(k+1)}$ may be reorganized modulo $\simeq$ into a derivation ψ' such that $\sigma(\psi') = R'_k; R'_{k+1}$ as follows: let p be the position of this occurrence of g in P and assume $P|_p = g(N_1, \ldots, N_m)$ and suppose $r_{\rho(k)}$ occurs in N_{l_1} and the head symbol of $r_{\rho(k+1)}$ occurs in N_{l_2} for $l_1, l_2 \in 1..m$ and $l_1 \neq l_2$:

1. First contract all the redexes in $r_i; \ldots; r_j$ prefixed by $p.l_2$
2. Second contract $r_{\rho(k+1)}$,
3. Third contract the (unique) residual of $r_{\rho(k)}$,
4. Finally contract the remaining redexes of $r_i; \ldots; r_j$, i.e. those prefixed by $p.1, \ldots, p.l_2 - 1, p.l_2 + 1, \ldots, p.m$.

Stage 3 (Irreversible Steps)

Finally, we show that also irreversible standardisation steps in $\mathcal{R}$ may be mimicked in the implementation: every irreversible standardisation step $\sigma(\chi) \stackrel{\mathtt{i}}{\Rightarrow} \omega$ in $\mathcal{R}$ may be mirrored as a non-empty series of standardisation steps $\chi \stackrel{\mathtt{r}}{\Rightarrow} \ldots \stackrel{\mathtt{r}}{\Rightarrow} \phi' \stackrel{\mathtt{i}}{\Rightarrow} \ldots \stackrel{\mathtt{i}}{\Rightarrow} \phi$ with at least one irreversible step in $\mathcal{R}_\sigma^{\mathsf{ES}}$, where $\sigma(\phi) = \omega$.

Hence the proof of Prop. 2 concludes by reasoning by contradiction since every standardisation step acting on the projected HO rewrite derivation may be mimicked by projection-related standardisation steps of the same nature (reversible/irreversible) over derivations in the implementation.

Suppose two $\mathcal{R}$-redexes $R_k : \sigma(P) \to \sigma(Q)$ and R_{k+1} can be permuted using an irreversible tile $R_k; R_{k+1} \rhd R'_k; \psi$. Observe the following:

Observation: Let r be a redex in M at some position p, instance of a rule $L \to R$. The *pattern* of r is the subterm of M at position p where the arguments of r (i.e. the terms substituted for the variables of L) are replaced by holes. Similarly to Rem. 2, all the symbols in the pattern of (the σ-ancestor of) R_{k+1} strictly above R_k in $\sigma(P)$ are present in P above the occurrence of $r_{\rho(k)}$. Moreover, none of these symbols occurs embraced by a substitution operator.

This follows from two facts:

1. first, by Lemma 1, the derivation $r_i; \ldots; r_j$ preserves all these symbols (in particular the lowest one), and
2. second, $r_{\rho(k+1)}$ is an $\mathcal{R}^{\mathsf{ES}}$-redex for $\mathcal{R}$ a $PERS_{db}$ (cf. Rem. 1) hence its *LHS* contains no occurrences of the substitution operator $\bullet[\bullet]$.

We consider two cases for Stage 2, reasoning by contradiction in each one: (A) The redex $r_{\rho(k)}$ in P occurs under an uncontributable symbol g belonging to the pattern of R_{k+1}, (B) All symbols above $r_{\rho(k)}$ in P belonging to the pattern of R_{k+1} are contributable. Note that the second case is not possible in the lambda calculus since the β-redex pattern always has the uncontributable symbol λ.

A. By Lemma 1(1) the derivation $r_i; \ldots; r_n$ preserves every uncontributable symbol strictly above $r_{\rho(k)}$. Among these symbols is the symbol g involved in the pattern of R_{k+1}. The redex $r_{\rho(k+1)}$ is above the position of this symbol. We reach a contradiction.
B. Suppose that the two $\mathcal{R}$-redexes R_k and R_{k+1} can be permuted using an irreversible tile $R_k; R_{k+1} \rhd R'_k; \psi$; we shall arrive at a contradiction. Let p be the occurrence of the unique σ-ancestor of the head symbol g of R_{k+1} in P, and $P|_p = g(M_1, .., M_m)$. By Lemma 1(2) the derivation $r_i; ...; r_j$ preserves p. Let $l \in 1..m$ such that $r_{\rho(k)}$ occurs in M_l. We may then reorganize modulo $\simeq$ the derivation $\psi = r_{\rho(k)}; r_i; \ldots; r_j; r_{\rho(k+1)}$ obtaining ψ', as follows:
   (a) First rewrite all redexes in $r_i; \ldots; r_j$ prefixed by $p.1$,$p.2$,...,$p.l - 1$,$p.l + 1$,..,$p.m$ in turn (i.e. first all those prefixed by $p.1$, then those by $p.2$, and so on) and those disjoint to p.
   (b) Second, rewrite all redexes prefixed by $p.l$ but disjoint to the (unique) residual of $r_{\rho(k)}$. At this moment the redex $r_{\rho(k+1)}$ must have emerged since

- by the Observation there are no substitution symbols in M_l between p and the position of $r_{\rho(k)}$, and
- $r_{j+1} = r_{\rho(k+1)}$, it is an $\mathcal{R}^{\mathsf{ES}}$-redex and hence its *LHS* contains no occurrences of the substitution operator $\bullet[\bullet]$.

(c) Thirdly, rewrite the (unique) residual of $r_{\rho(k)}$, say $r'_{\rho(k)}$, followed by the redexes in $r_i; \ldots; r_j$ prefixed by the occurrence of $r'_{\rho(k)}$, i.e. those not rewritten in Step 1 or Step 2.

(d) Finally, rewrite $r_{\rho(k+1)}$.

Note that $\psi' = \psi'_1; \psi'_2$ where ψ'_1 consists solely of σ-rewrite steps (see Step 1 and 2, above) and $\psi'_2 = r'_{\rho(k)}; r_{k_1}; \ldots; r_{k_m}; r_{\rho(k+1)}$ for some $m \leq j - i$. Applying $m + 1$ irreversible standardisation steps starting from ψ'_2 we may obtain $\psi'_3 = r'_{\rho(k+1)}; \psi_{r_{\rho(k)}}; \psi_{r_{k_1}}; \ldots; \psi_{r_{k_m}}$. Finally, setting $\psi = \psi'_1; \psi'_3$ we may conclude.

This concludes the proof of Prop. 2. As a consequence we have:

Theorem 2. *Let $\mathcal{R}$ be any orthogonal pattern ERS_{db}. All needed derivations normalise in the ES-based implementation $\mathcal{R}^{\mathsf{ES}}_{\sigma}$ of $\mathcal{R}$.*

Remark 3. Although our interest is in normalisation we would like to point out that Prop. 2 may be seen as reducing standardisation for HORS to that of first-order systems. Given a derivation $\chi : M \twoheadrightarrow N$ in an orthogonal pattern ERS_{db} $\mathcal{R}$, we recast χ in the ES-based implementation of $\mathcal{R}$, then we standardise the resulting derivation in the first-order setting [Bou85] and finally we project back to the HO-setting. The resulting $\mathcal{R}$ derivation ϕ shall not be just any standard derivation from M to N, but also Lévy-permutation equivalent to χ, in other words, $\phi \equiv \chi$. This may be verified by proving that $\varphi \Rightarrow \phi$ implies $\sigma(\varphi) \equiv \sigma(\phi)$.

5 Conclusions

We have addressed normalisation by needed reduction in the ES-based approach to the implementation of HORS. Melliès [Mel95] observed that the implementation of a higher-order rewrite system by means of calculi of explicit substitution may change its normalisation properties fundamentally; indeed a term possessing no infinite derivations in the λ-calculus may lose this property when shifting to the $\lambda\sigma$-calculus. Based on an extension of the theory of needed redexes to overlapping systems [Mel00] we have shown that all needed derivations normalise in the ES-based implementation of any orthogonal pattern HORS; the latter result has been established in the setting of the ERS_{db} formalism for higher-order rewriting. The key property that has been addressed in order to apply the aforementioned theory is to show that standard derivations in the ES-based implementation of a HORS project to standard derivations in the higher-order setting (Std-Projection Proposition). The fact that this key property is all that is required owes to a simplified proof of [Mel00, Thm.7.1] (Prop. 3).

In [BKR01] the ES-based implementation of HORS does not fix a particular calculus of explicit substitutions. Instead a macro-based presentation encompassing a wide class of calculi of ES is used. The study of the abstract properties that make the proof of the Std-Projection Proposition go through would allow the results presented here to be made independent of σ, the calculus of ES which we have dealt with in this paper.

Further in this line, it would be interesting to insert the work presented here into the axiomatic setting of Axiomatic Rewrite Systems as developed in [Mel96, Mel00]. This would require a formulation of calculi of ES based on abstract axiomatic properties, work which we are currently undertaking.

Acknowledgements. To Delia Kesner, Alejandro Ríos and the referees for providing valuable suggestions.

References

[ACCL91] M. Abadi, L. Cardelli, P-L. Curien, and J-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 4(1):375–416, 1991.

[Bar84] H.P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, revised edition, 1984.

[BBLRD96] Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, 1996.

[BKR00] E. Bonelli, D. Kesner, and A. Ríos. A de Bruijn notation for Higher-Order Rewriting. In *Proceedings of the 11th RTA*, number 1833 in LNCS. Springer-Verlag, 2000.

[BKR01] E. Bonelli, D. Kesner, and A. Ríos. From Higher-Order to First-Order Rewriting. In *Proceedings of the 12th RTA*, number 2051 in LNCS. Springer-Verlag, 2001.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Bon01] E. Bonelli. *Term rewriting and explicit substitutions*. PhD thesis, Université de Paris Sud, November 2001.

[Bon03] E. Bonelli. A normalisation result for higher-order calculi with explicit substitutions, 2003. Full version of this paper. `http://www-lifia.info.unlp.edu.ar/~ eduardo/`.

[Bou85] G. Boudol. Computational semantics of term rewrite systems. In M. Nivat and J.C. Reynolds, editors, *Algebraic methods in Semantics*. Cambridge University Press, 1985.

[CHR92] P-L. Curien, T. Hardin, and A. Ríos. Strong normalization of substitutions. In *Proceedings of Mathematical Foundations of Computer Science*, number 629 in LNCS, pages 209–217. Springer-Verlag, 1992.

[dB72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation with application to the church-rosser theorem. *Indag. Mat.*, 5(35), 1972.

[DG01] R. David and B. Guillaume. A λ-calculus with explicit weakening and explicit substitutions. *Mathematical Structures in Computer Science*, 11(1), 2001.

[DJ90] N. Dershowitz and J-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–309. North-Holland, 1990.

[GKK00] J. Glauert, R. Kennaway, and Z. Khasidashvili. Stable results and relative normalization. *Journal of Logic and Computation*, 10(3), 2000.

[HL91] G. Huet and J-J. Lévy. Computations in orthogonal rewriting systems. In J.L. Lassez and G.D. Plotkin, editors, *Computational Logic; Essays in honor of Alan Robinson*, pages 394–443. MIT Press, 1991.

[HMP96] Th. Hardin, L. Maranget, and P. Pagano. Functional back-ends within the lambda-sigma calculus. In *Proceedings of the International Conference on Functional Programming*, LNCS. Springer-Verlag, 1996.

[Klo80] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, CWI, Amsterdam, 1980. Mathematical Centre Tracts n.127.

[Klo92] J.W. Klop. Term rewriting systems. *Handbook of Logic in Computer Science*, 2:1–116, 1992.

[KOvR93] J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory Reduction Systems: introduction and survey. *Theoretical Computer Science*, 121(1–2):279–308, 1993.

[KR95] F. Kamareddine and A. Ríos. A λ-calculus à la de Bruijn with explicit substitutions. In *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, number 982 in LNCS. Springer Verlag, 1995.

[Lév78] J-J. Lévy. *Réductions correctes et optimales dans le lambda-calcul*. PhD thesis, Université Paris VII, 1978.

[Mar92] L. Maranget. *La stratégie paresseuse*. PhD thesis, Université Paris VII, 1992.

[Mel95] P-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate. In *Proceedings of Typed Lambda Calculi and Applications*, number 902 in LNCS. Springer-Verlag, 1995.

[Mel96] P-A. Melliès. *Description Abstraite des Systèmes de Réécriture*. PhD thesis, Université Paris VII, 1996.

[Mel00] P-A. Melliès. Axiomatic Rewriting Theory II: The $\lambda\sigma$-calculus enjoys finite normalisation cones. *Journal of Logic and Computation*, 10(3):461–487, 2000.

[Mil91] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In P. Schroeder-Heister, editor, *Proceedings of the International Workshop on Extensions of Logic Programming, FRG, 1989*, number 475 in Lecture Notes in Artificial Intelligence. Springer-Verlag, December 1991.

[Nip91] T. Nipkow. Higher-order critical pairs. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, July 1991.

[Oos94] V. van Oostrom. *Confluence for Abstract and Higher-order Rewriting*. PhD thesis, Vrije University, 1994.

[Oos99] V. van Oostrom. Normalization in weakly orthogonal rewriting. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, number 1631 in LNCS. Springer-Verlag, 1999.

When Ambients Cannot Be Opened*

Iovka Boneva and Jean-Marc Talbot

Laboratoire d'Informatique Fondamentale de Lille, France
{boneva,talbot}@lifl.fr

Abstract. We investigate expressiveness of a fragment of the ambient calculus, a formalism for describing distributed and mobile computations. More precisely, we study expressiveness of the pure and public ambient calculus from which the capability `open` has been removed, in terms of the reachability problem of the reduction relation. Surprisingly, we show that even for this very restricted fragment, the reachability problem is not decidable. At a second step, for a slightly weaker reduction relation, we prove that reachability can be decided by reducing this problem to markings reachability for Petri nets. Finally, we show that the name-convergence problem as well as the model-checking problem turn out to be undecidable for both the original and the weaker reduction relation.

1 Introduction

The ambient calculus [5] is a formalism for describing distributed and mobile computation in terms of *ambients*, named collections of running processes and nested sub-ambients. A state of computation has a tree structure induced by ambient nesting. Mobility is represented by re-arrangement of this tree (an ambient may move inside or outside other ambients) or by deletion of a part of this tree (a process may dissolve the boundary of some ambient, revealing its contents). Mobility is ruled by the capabilities `in`, `out`, `open` owned by the ambients. The ambient calculus also inherits replication, name restriction and asynchronous communication from the π-calculus [16].

The ambient calculus is a very expressive formalism. It has been proved Turing-complete in [5] by encoding Turing-machine computations. This encoding uses both mobility features from the calculus as well as name restriction. However, several variants of the ambient calculus have been proposed so far [14,2,19] by adding and/or retracting features from the original calculus. In [14], the safe ambient calculus introduces some co-capabilities. They are used as an agreement on mobility between the moving ambient (executing a capability) and the ambient where it will move to (executing the corresponding co-capability). The boxed ambient calculus is another variant [2]; in this calculus, the possibility to dissolve boundary of ambients has disappeared and is replaced by a more sophisticated communication mechanism.

Studying precise expressiveness of these different variants of the ambient calculus is crucial as it may separate necessary features from redundant ones and it may also help to design or improve algorithms to verify [18,7,6] or analyze [11,9] programs written in these ambient formalisms.

* The authors are grateful to S. Tison and Y. Roos for fruitful discussions and thank the anonymous referees for valuable comments. This work is supported by an ATIP grant from CNRS.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 169–184, 2003.

Some works aimed already to study expressiveness of ambient calculus: in [20], it is shown that the π-calculus, a formalism based only on name communication, can be simulated in the communication-free safe ambient calculus. In [7], the pure and public ambient calculus (an ambient calculus in which communication and name restriction are omitted) is considered and proved to be still very powerful: for this restricted fragment, the reachability problem (*i.e.* given two processes P and Q, can the process P evolve into the process Q ?) can not be decided. Recently, in two different works [13] and [3], it has been established that this fragment is actually Turing-complete. In [3], the authors also showed that the ambient calculus limited to `open` capabilities and name restriction is Turing-complete as it can simulate counters machines computations [17]. The name restriction is needed there as if omitted, divergence for reductions of processes can be decided. In this latter paper, the following question is raised: what is the expressiveness power of the "dual" calculus, a calculus in which the `open` capability is dropped whereas the `in`, `out` capabilities are preserved ?

In this paper, we investigate expressiveness of pure and public mobile ambients without the `open` capability. Hence, the reduction of a process is limited to the rearrangement of its tree structure. To be more precise, we study the reachability problem for such ambient processes. We show that for this calculus reachability for the reduction relation between two processes can not be decided. To prove this result, we use a non-trivial reduction to the acceptance problem of two-counters machines [17]. We figured out that the major source of undecidability for this fragment comes from the definition of replication as part of the structural congruence relation (the structural congruence aims to identify equivalent processes having different syntactic representations). Indeed, we show that if this definition of replication is presented as an (oriented) reduction rule then the reachability for the reduction relation between two processes becomes decidable. We prove this statement by reducing this problem to the reachability problem of markings in Petri nets [15]. Finally, we investigate two problems related to reachability. The first problem is the name-convergence problem [10]: a process converges to some name n if this process can be reduced to some process having an ambient named n at its top-level. We show that this problem is undecidable however the definition of replication is presented. The second problem is the model-checking problem against the ambient logic [4]. It is easy to show that the name-convergence problem can be reduced to an instance of the model-checking problem. Thus, this latter is undecidable as well.

The paper is organized as follows: in Section 2, we give definitions for the ambient calculus we consider here. Section 3 is devoted to the reachability problem for this fragment. We give there a negative result: this problem is undecidable. In Section 4, we consider a weak calculus based on a different reduction relation; we show that for this particular case the reachability problem becomes decidable. Finally, in Section 5, we consider other problems such as model-checking and name-convergence. We show that for the two kinds of reduction we considered those problems are not decidable.

2 Definitions

We present in this section the fragment of the ambient calculus we consider all along this paper. This fragment corresponds to the ambient calculus defined in [5] for which

both name restriction, communication and the `open` capability have been dropped. We call this fragment `in/out` ambient calculus.

We assume countably many *names* ranging over by $n, m, a, b, c, \ldots$. For any name n, we consider *capabilities* α of the form $\texttt{in}\ n$ and $\texttt{out}\ n$. The following table defines the syntax of *processes* of our calculus.

Processes:

$P, Q, R ::=$	processes		
$\mathbf{0}$	inactivity	$P \mid Q$	composition
$n[P]$	ambient	$\alpha.P$	action prefix
$!P$	replication		

The semantics of our calculus is given by two relations. The *reduction relation* $P \rightarrow Q$ describes the evolution of processes over time. We write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$. The *structural congruence* relation $P \equiv Q$ relates different syntactic representations of the same process; it is used to define the reduction relation.

The structural congruence is defined as the least relation over processes satisfying the axioms from the table below:

Structural Congruence $P \equiv Q$:

$P \equiv P$	(Str Refl)	$P \mid \mathbf{0} \equiv P$	(Str Par Zero)
$P \equiv Q \Rightarrow Q \equiv P$	(Str Symm)	$P \mid Q \equiv Q \mid P$	(Str Par Comm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Str Trans)	$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Str Par Assoc)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Str Par)	$!P \equiv !P \mid P$	(Str Repl Copy)
$P \equiv Q \Rightarrow n[P] \equiv n[Q]$	(Str Amb)	$!\mathbf{0} \equiv \mathbf{0}$	(Str Repl Zero)
$P \equiv Q \Rightarrow \alpha.P \equiv \alpha.Q$	(Str Action)	$!!P \equiv !P$	(Str Repl Repl)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Str Repl)	$!(P \mid Q) \equiv !P \mid !Q$	(Str Repl Par)

The first column specifies that $\equiv$ is a congruence relation over processes. The second one specifies properties of the replication and parallel operators: in particular, it states that the parallel operator is associative-commutative and has $\mathbf{0}$ as neutral element.

The reduction relation is defined as the least relation over processes satisfying the following set of axioms:

Reduction: $P \rightarrow Q$

$n[\texttt{in}\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$	(Red In)
$m[n[\texttt{out}\ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$	(Red Out)
$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$	(Red Par)
$P \rightarrow Q \Rightarrow n[P] \rightarrow n[Q]$	(Red Amb)
$P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$	(Red $\equiv$)

When writing processes, we may omit irrelevant occurrences of the inactive process $\mathbf{0}$. For instance, we may write $n[]$ for $n[\mathbf{0}]$ and $\texttt{in}\ a.\texttt{out}\ b$ for $\texttt{in}\ a.\texttt{out}\ b.\mathbf{0}$.

3 The Reachability Problem for `in/out` Ambient Calculus

In this section we investigate the reachability problem for `in/out` ambient calculus: "Given two processes P, Q, does $P \rightarrow^* Q$ hold ?". We show that, despite the small

fragment of the ambient calculus we consider, this problem is undecidable by defining a reduction to the acceptance problem of two-counters machine [17].

A *two-counters machine* $\mathcal{M}$ is given by a four tuple $(\mathcal{Q}, q_i, q_f, \Delta)$ where $\mathcal{Q}$ is a finite set of states, $q_i \in \mathcal{Q}$ is the initial state, $q_f \in \mathcal{Q}$ is the final state; Δ is a finite subset of $(\mathcal{Q} \times \{+, -, =\} \times \{0, 1\} \times \mathcal{Q})$ called the *transition relation*. A *configuration* of the machine $\mathcal{M}$ is given by a triple (q, c_0, c_1) belonging to the set $(\mathcal{Q}, \mathbb{N}, \mathbb{N})$ (where $\mathbb{N}$ is the set of natural numbers); c_0, c_1 are the two counters. The transition relation Δ defines a *step relation* $\vdash_{\mathcal{M}}$ over configurations as follows: $(q, c_0, c_1) \vdash_{\mathcal{M}} (q', c_0', c_1')$ iff one of the three statements is true for $i, j \in \{0, 1\}$ and $i \neq j$: (**i**) $(q, =, i, q')$ in Δ, $c_i = c_i' = 0$ and $c_j = c_j'$, (**ii**) $(q, +, i, q')$ in Δ, $c_i' = c_i + 1$ and $c_j = c_j'$, (**iii**) $(q, -, i, q')$ in Δ, $c_i > 0$, $c_i' = c_i - 1$ and $c_j = c_j'$.

Let $\vdash_{\mathcal{M}}^*$ be the reflexive and transitive closure of $\vdash_{\mathcal{M}}$. A two-counters machine *accepts* a natural v if $(q_i, v, 0) \vdash_{\mathcal{M}}^* (q_f, 0, 0)$.

Theorem 1. *[17] For an arbitrary two-counters machine $\mathcal{M}$ and an arbitrary natural v, it is undecidable to test whether $\mathcal{M}$ accepts v.*

We express the acceptance problem of a natural v by a machine $\mathcal{M}$ in terms of an instance of the reachability problem in the $\mathtt{in}/\mathtt{out}$ ambient calculus. We encode within a process $[\![(q, v_0, v_1)]\!]$ the configuration of the machine (q, v_0, v_1) (the current state and the values for the two counters) and the transition relation Δ of the machine. The step relation of the machine is then simulated by the reduction of this process. It should be noticed that not all the different reductions that could be performed by the process indeed participate to the step relation; the process may engage some wrong reduction steps. However, we show that in this case, the process either goes stuck in a form that does not correspond to some valid representation of the machine or carries on some reduction steps but without any possibility to resume into an encoding of the two-counters machine. Let us now describe the process $[\![(q, v_0, v_1)]\!]$: we assume for any state q occurring in Δ, two ambient names q and q_t: q represents the state of the machine and q_t is used to denote a possible transition of the machine being in the state q. The process $[\![(q, v_0, v_1)]\!]$ is defined as

$$q[\mathtt{in}\, q_t] \mid c_0[V(v_0) \mid !k[\mathtt{out}\, c_0]] \mid c_1[V(v_1) \mid !k[\mathtt{out}\, c_1]] \mid [\![\Delta]\!] \mid P_G$$

The ambient $q[\mathtt{in}\, q_t]$ represents the current state of the machine; the ambients c_0 and c_1 represent the counters with their respective values $V(v_0)$ and $V(v_1)$. The parametrized process $V(v)$ encodes the value v recursively as: (i) $V(0) = n[!I \mid !D \mid \mathtt{in}\, k \mid a[\mathbf{0}]]$ and (ii) $V(v+1) = n[!I \mid !D \mid \mathtt{in}\, k \mid b[\mathbf{0}] \mid V(v)]$. Intuitively, the value v of the counter is given by the number of ambients n in the process $V(v)$ minus 1. The two processes I and D are defined as $I = \mathtt{in}\, i.\mathtt{in}\, n.\mathbf{0}$, $D = \mathtt{in}\, z.\mathtt{in}\, z'.\mathtt{out}\, z'.\mathtt{out}\, n.\mathbf{0}$: the process I is used to increment the counter and D to decrement it.

The process $[\![\Delta]\!]$ represents the transition rules of the machine and is defined as the parallel composition of the replicated processes encoding each transition rule. Formally, we have inductively $[\![\varnothing]\!] = \mathbf{0}$ and $[\![\Delta \cup \{(q, s, j, q')\}]\!] = [\![\Delta \setminus \{(q, s, j, q')\}]\!] \mid ![\![(q, s, j, q')]\!]$. For each kind of transition rules:

- $[\![(q,=,j,q')]\!] = q_t[q'[\beta_j^q.(\texttt{in}\ n.\texttt{in}\ a.\texttt{out}\ a.\texttt{out}\ n.\texttt{out}\ c_j.\texttt{in}\ q_t')]]$
- $[\![(q,+,j,q')]\!] = q_t[i[\beta_j^q.N_{q'} \mid \texttt{in}\ q'.\texttt{out}\ q'.\texttt{out}\ c_j]]$ with $N_{q'} = n[\texttt{out}\ i.(!I \mid !D \mid \texttt{in}\ k \mid b[\mathbf{0}] \mid q'[\delta_{q'}.\texttt{in}\ q_t'])]$ and $\delta_{q'} = \texttt{in}\ n.\texttt{out}\ n.\texttt{out}\ n.\texttt{in}\ i.\texttt{out}\ i.$
- $[\![(q,-,j,q')]\!] = q_t[d[\beta_j^q.\texttt{in}\ n.\texttt{in}\ b.\texttt{out}\ b.Z_q']]$ with $Z_{q'} = z[\texttt{out}\ d.z'[\texttt{out}\ z.q'[\texttt{out}\ z'.\texttt{out}\ n.\texttt{out}\ k.\texttt{in}\ q_t']]]$

where β_j^q is defined as the sequence of capabilities $\texttt{in}\ q.\texttt{out}\ q.\texttt{out}\ q_t.\texttt{in}\ c_j$.

Finally, the process P_G plays the role of some garbage collector (using that $!P \mid P$ is structurally congruent to $!P$): assuming that $q^1, \ldots, q^l$ are the states of the two-counters machine, the process P_G is defined as

$$!k[\mathbf{0}] \mid !i[\mathbf{0}] \mid !k[n[!I \mid !D \mid d[\mathbf{0}] \mid z[\mathbf{0}] \mid z'[\mathbf{0}] \mid b[\mathbf{0}]]] \mid !q_t^1[q^1[\mathbf{0}]] \mid \ldots \mid !q_t^l[q^l[\mathbf{0}]]$$

Note that at any step of the computation, because of the two subprocesses $!k[\texttt{out}\ c_0]$ and $!k[\texttt{out}\ c_1]$ contained respectively in the ambients c_0 and c_1, the process can always perform a reduction step: a copy of $k[\texttt{out}\ c_j]$ can "jump" outside of the counter c_j. However, the resulting process $k[\mathbf{0}]$ is simply garbage-collected by the process P_G. Hence, the process would simply reduce into itself; therefore, we will not take any longer into account these irrelevant reduction steps.

Let us now describe how the process $[\![(q, v_0, v_1)]\!]$ may evolve into another process $[\![(q', v_0', v_1')]\!]$ according to the transition rules of the two-counters machine. First, the ambient $q[\texttt{in}\ q_t]$ reduces with a sibling ambient q_t. Note that a misleading reduction with an ambient $q_t[q[\mathbf{0}]]$ from the process P_G may happen. If so, the computation is stuck. If there exists a transition from the state q in the two-counter machine, then an alternative reduction could occur with an ambient q_t provided by Δ leading to

$$q_t[q[\mathbf{0}] \mid \eta[\beta_j^q.\ldots. \mid \ldots]] \mid c_0[\ldots] \mid c_1[\ldots] \mid [\![\Delta]\!] \mid P_G \qquad \text{with } \eta \in \{i, d, q'\}$$

The sequence of capabilities β_j^q allows the transition which has the "control" (*i.e.* q_t contains $q[\mathbf{0}]$) to provide "material" (represented as the ambient η) to treat the counter addressed by this transition. Once, β_j^q is executed we obtain (assuming the transition addresses the counter c_0) $c_0[\eta[\ldots] \mid \ldots] \mid c_1[\ldots] \mid [\![\Delta]\!] \mid P_G$ with $\eta \in \{i, d, q'\}$.

Note that $q_t[q[\mathbf{0}]]$ remaining at the top-level is garbage collected by P_G. Now, the reductions differ according to the kind of transition that was chosen (assuming the transition addresses the counter c_0, things being similar for c_1):

• **for** $(\mathbf{q}, =, \mathbf{0}, \mathbf{q}')$**:** the ambient $\eta[\ldots]$ is $q'[\texttt{in}\ n.\texttt{in}\ a.\texttt{out}\ a.\texttt{out}\ n.\texttt{out}\ c_0.\texttt{in}\ q_t']$. If the value of c_0 is 0 then c_0 contains an ambient $n[a[\mathbf{0}] \mid \ldots]$. The sequence of capabilities $\texttt{in}\, n.\texttt{in}\, a.\texttt{out}\, a.\texttt{out}\, n.\texttt{out}\, c_0$ can be executed and so, the next configuration is obtained. Note that if $V(v_0)$ is not 0, then the process remains stuck as q can not execute its capability $\texttt{in}\ a$.

• **for** $(\mathbf{q}, +, \mathbf{0}, \mathbf{q}')$**:** the ambient $\eta[\ldots]$ is $i[N_{q'} \mid \texttt{in}\ q'.\texttt{out}\ q'.\texttt{out}\ c_0]]$ and the value $V(v_0)$ of the form $n[!I \mid \ldots]$ is one if its siblings. Reminding that $N_{q'} = n[\texttt{out}\ i.(!I \mid !D \mid \texttt{in}\, k \mid b[\mathbf{0}] \mid q'[\delta_{q'}.\texttt{in}\, q_t'])]$, we can see that $N_{q'}$ contains roughly an ambient n used for incrementing and the successor state q' that will try to check that the incrementing has been done properly.

$V(v_0)$ executes the sequence $I = \texttt{in}\ i.\texttt{in}\ n$ leading to an ambient c_0

$$c_0\left[i\left[\begin{array}{l} n[\texttt{out}\ i.(!I \mid !D \mid \texttt{in}\ k \mid b[\mathbf{0}] \mid q'[\delta_{q'}.\texttt{in}\ q'_t]) \mid V(v_0)] \\ \mid \texttt{in}\ q'.\texttt{out}\ q'.\texttt{out}\ c_0 \end{array}\right] \mid !k[\texttt{out}\ c_0]\right]$$

By executing $\texttt{out}\ i$ from the top ambient n, one obtains

$$c_0[i[\texttt{in}\ q'.\texttt{out}\ q'.\texttt{out}\ c_0] \mid n[V(v_0) \mid !I \mid !D \mid \texttt{in}\ k \mid b[\mathbf{0}] \mid q'[\delta_{q'}.\texttt{in}\ q'_t]] \mid !k[\texttt{out}\ c_0]]$$

At that point, we have almost $V(v_0+1)$ except the presence of q' in the top ambient n. Then, by using $\delta_{q'} = \texttt{in}\ n.\texttt{out}\ n.\texttt{out}\ n.\texttt{in}\ i.\texttt{out}\ i$, q' notices that it has $V(v_0)$ as a sibling (by executing $\texttt{in}\ n.\texttt{out}\ n$) and goes outside of n.

$$c_0[i[\texttt{in}\ q'.\texttt{out}\ q'.\texttt{out}\ c_0] \mid q'[\texttt{in}\ i.\texttt{out}\ i.\texttt{in}\ q'_t] \mid V(v_0+1) \mid !k[\texttt{out}\ c_0]]$$

The ambient i detects it has a sibling q' (by executing $\texttt{in}\ q'.\texttt{out}\ q'$) and the ambient q' enters i yielding $c_0[i[\texttt{out}\ c_0 \mid q'[\texttt{out}\ i.\texttt{in}\ q'_t]] \mid V(v_0+1) \mid !k[\texttt{out}\ c_0]]$. Then i goes outside of c_0. So, the process is

$$i[q'[\texttt{out}\ i.\texttt{in}\ q'_t]] \mid c_0[V(v_0+1) \mid \ldots] \mid c_1[V(v_1) \mid \ldots] \mid [\![\Delta]\!] \mid P_G$$

The ambient q' exits i, producing the process $i[\mathbf{0}] \mid q'[\texttt{in}q'_t]$. The process $i[\mathbf{0}]$ is garbage-collected by P_G, and the result indeed corresponds to the process $[\![(q', v_0+1, v_1)]\!]$.

• **for** $(\mathbf{q}, -, \mathbf{0}, \mathbf{q}')$**:** the ambient $\eta[\ldots]$ is $d[\texttt{in}\ n.\texttt{in}\ b.\texttt{out}\ b.Z_{q'}]$. If the value of $V(v_0)$ is strictly positive then it is of the form $n[b[\mathbf{0}] \mid \ldots]$. Then d executes $\texttt{in}\ n.\texttt{in}\ b.\texttt{out}\ b$; the contents of c_0 is then $c_0[n[d[Z_{q'}] \mid V(v_0-1) \mid !I \mid !D \mid \texttt{in}\ k \mid b[\mathbf{0}]] \mid !k[\texttt{out}\ c_0]]$.

Note that if $V(v_0)$ is 0, then the process remains stuck as d can not execute its capability $\texttt{in}\ b$. The role of $Z_{q'}$ is interact with $V(v_0-1)$ in order to trigger for this latter the possibility to go outside of its surrounding ambient n. We recall that $Z_{q'} = z[\texttt{out}\ d.z'[\texttt{out}\ z.q'[\texttt{out}\ z'.\texttt{out}\ n.\texttt{out}\ k.\texttt{in}\ q'_t]]]$ and that $V(v_0-1)$ contains at its top-level $D = \texttt{in}\ z.\texttt{in}\ z'.\texttt{out}\ z'.\texttt{out}\ n.\mathbf{0}$. The ambient z from $Z_{q'}$ exits d; then, the ambient $V(v_0-1)$ executes the capabilities $\texttt{in}\ z.\texttt{in}\ z'$ from D and finally, z' leaves z. We reach the following situation for the ambient z':

$$z'[q'[\texttt{out}\ z'.\texttt{out}\ n.\texttt{out}\ k.\texttt{in}\ q'_t] \mid n[\texttt{out}\ z'.\texttt{out}\ n \mid \texttt{in}\ k \mid \ldots]]$$

The ambient n executes the remaining capabilities from D, that is $\texttt{out}\ z'.\texttt{out}\ n$; concurrently, the ambient q' exits z'. The contents of c_0 is then

$$!k[\texttt{out}\ c_0] \mid V(v_0-1) \mid n[\texttt{in}\ k \mid !I \mid !D \mid b[] \mid d[] \mid z[] \mid z'[] \mid q'[\texttt{out}\ n.\texttt{out}\ k.\texttt{in}\ q'_t]]$$

At that point, the value of the counter has been decremented; the rest of the computation aims to "clean up" the process allowing the computation to carry on. The ambient n moves inside an ambient $k[\texttt{out}\ c_0]$. So, we have in c_0

$$!k[\texttt{out}\ c_0] \mid V(v_0-1) \mid k[\texttt{out}c_0 \mid n[!I \mid !D \mid b[] \mid d[] \mid z[] \mid z'[] \mid q'[\texttt{out}n.\texttt{out}k.\texttt{in}q'_t]]]$$

In some arbitrary order, the ambient k (containing n) leaves the counter c_0 and q' leaves the ambient n.

$$k[n[!I \mid !D \mid b[] \mid d[] \mid z[] \mid z'[]] \mid q'[\texttt{out}\ k.\texttt{in}\ q'_t]] \mid c_0[V(v_0-1) \mid ..] \mid c_1[V(v_1) \mid ..] \mid [\![\Delta]\!] \mid P_G$$

Finally, the ambient q' exits k by executing its capability $\texttt{out}\ k$ and the subprocess $k[n[!I \mid !D \mid b[\mathbf{0}] \mid d[\mathbf{0}] \mid z[\mathbf{0}] \mid z'[\mathbf{0}]]]$ is garbage-collected by the process P_G. The result indeed corresponds to the expected process $[\![(q', v_0 - 1, v_1)]\!]$.

We described above how the step relation of the two-counters machine can be simulated by some reductions of a process encoding some configuration. The sequences of reductions we described for each kind of transition relations are not the only possible ones for the process we considered. However, following some different sequences of reduction would either lead to a stuck process or would produce only processes that do not correspond to an encoding of the two-counters machine[1]. Our encoding is correct in the following sense:

Proposition 1. *For any two-counters machine $\mathcal{M} = (\mathcal{Q}, q_i, q_f, \Delta)$ and any arbitrary natural v, $\mathcal{M}$ accepts v iff $[\![(q_i, v, 0)]\!] \rightarrow^* [\![(q_f, 0, 0)]\!]$.*

Hence, as an immediate consequence using Theorem 1:

Theorem 2. *For any two arbitrary processes P, Q from the* $\texttt{in}/\texttt{out}$ *ambient calculus, it is undecidable to test whether $P \rightarrow^* Q$.*

We believe that our encoding can easily be adapted to safe mobile ambients [14] from which name restriction, communication, the capability $\texttt{open}$ and the co-capability $\overline{\texttt{open}}$ have been removed.

We claim that one of the sources of undecidability of the reachability problem is the rule (Str Repl Copy) from the structural congruence. On one hand, this rule can be used to exhibit a new process P from $!P$; this creates new possible interactions through reduction for this occurrence of P. On the other hand, it can be used to transform $!P \mid P$ into $!P$; we used this feature in our encoding to provide a garbage-collecting mechanism. Supporting our claim, we will see in the next section that if we drop this second possibility, then the reachability problem becomes decidable.

4 The Reachability Problem for a Weaker Reduction Relation

In this section, we study the reachability problem for the $\texttt{in}/\texttt{out}$ ambient calculus equipped with a weaker reduction relation. We show that for this new reduction relation, the reachability problem becomes decidable.

4.1 Definitions

The weaker reduction relation we consider here has been introduced in [1] [2]. Its main feature is to turn the axiom defining replication, that is $!P \equiv !P \mid P$ (Str Repl Copy), into an (oriented) reduction rule $!P \rightarrow P \mid !P$ (wRed Repl).

We consider the *weak structural congruence* $\cong$ defined as the least congruence relation over processes satisfying all the axioms defining $\equiv$ except (Str Repl Copy). We

[1] We prove this fact by defining a general shape matching all reachable processes and by applying an exhaustive inspection of all possibles reductions.

[2] In [19], the iteration is also defined by means of a reduction rule, but for explicit recursion instead of replication.

called this structural congruence weak as obviously $P \cong Q$ implies $P \equiv Q$ whereas the converse does not hold. Based on this weak structural congruence, we define a *weak reduction relation* as the least relation satisfying the following axioms:

Weak Reduction: $P \rightarrow_{\mathrm{w}} Q$

$n[\mathtt{in}\ m.P \mid Q] \mid m[R] \rightarrow_{\mathrm{w}} m[n[P \mid Q] \mid R]$	(wRed In)
$m[n[\mathtt{out}\ m.P \mid Q] \mid R] \rightarrow_{\mathrm{w}} n[P \mid Q] \mid m[R]$	(wRed Out)
$!P \rightarrow_{\mathrm{w}} P \mid !P$	(wRed Repl)
$P \rightarrow_{\mathrm{w}} Q \Rightarrow P \mid R \rightarrow_{\mathrm{w}} Q \mid R$	(wRed Par)
$P \rightarrow_{\mathrm{w}} Q \Rightarrow n[P] \rightarrow_{\mathrm{w}} n[Q]$	(wRed Amb)
$P' \cong P, P \rightarrow_{\mathrm{w}} Q, Q \cong Q' \Rightarrow P' \rightarrow_{\mathrm{w}} Q'$	(wRed $\cong$)

This new reduction relation is strictly weaker than the one presented in Section 2:

Proposition 2. *For all processes P, Q if $P \rightarrow_w^* Q$ then $P \rightarrow^* Q$. Moreover, there exist two processes P' and Q' such that $P' \rightarrow^* Q'$ and $P' \not\rightarrow_w^* Q'$.*

Let us point out that if we consider additionally `open` capabilities and enrich the definition of $\rightarrow_{\mathrm{w}}$ with the rule $\mathtt{open}\ n.P \mid n[Q] \rightarrow_{\mathrm{w}} P \mid Q$ (Red Open) then the reachability problem for this weak reduction relation is undecidable: the encoding of the Post Correspondence Problem given in [7] provides a proof for this statement.

4.2 The Reachability Problem

We will show that the reachability problem is decidable for the weak reduction relation.

Theorem 3. *For all processes S and T, it is decidable to test whether $S \rightarrow_w^* T$.*

The rest of this section is devoted to the proof of Theorem 3. The main guidelines of this proof are as follows: first, we introduce a notion of normal form for processes for which we specialize the weak reduction relation; secondly, we show that the reachability problem for two arbitrary processes can be expressed as the reachability problem on their respective normal forms. Finally, we show how to reduce reachability problem for normal forms into markings reachability in Petri nets, a problem known to be decidable.

• **From weak reduction to weak reduction over normal forms:** As done in [12,8], we consider the axioms $P \mid \mathbf{0} \cong P$, $!\mathbf{0} \cong \mathbf{0}$, $!!P \cong !P$ and $!(P \mid Q) \cong !P \mid !Q$ from the definition of $\cong$. We turn those axioms into a rewrite system $\mathcal{W}$ by orienting them from left to right and we denote $\rightsquigarrow_{\mathcal{W}}$ the AC-rewrite relation induced by $\mathcal{W}$ (taking into account associativity and commutativity for the parallel operator $|$). It can be shown that the AC-rewrite relation $\rightsquigarrow_{\mathcal{W}}$ is terminating and confluent. Hence, for any process P, there exists a unique (modulo associativity and commutativity for $|$) normal form $\tilde{P}$ of P wrt $\rightsquigarrow_{\mathcal{W}}$. This implies also that $\cong$ is decidable.

In the sequel we denote $=_{\mathsf{AC}}$ the equality relation over processes modulo associativity and commutativity for the parallel operator $|$.

We introduce a new reduction relation for normal forms. In particular, we require that any process in normal form is reduced to some normalized process. This reduction relation is denoted $\twoheadrightarrow$ and is given in the table below:

Normal Weak Reduction: $P \twoheadrightarrow Q$

$n[\texttt{in}\, m.P] \mid m[\mathbf{0}] \twoheadrightarrow m[n[P]]$	(wRed In 1)
$n[\texttt{in}\, m.P] \mid m[R] \twoheadrightarrow m[n[P] \mid R]$ if $R \neq \mathbf{0}$	(wRed In 2)
$n[\texttt{in}\, m.\mathbf{0} \mid Q] \mid m[\mathbf{0}] \twoheadrightarrow m[n[Q]]$	(wRed In 3)
$n[\texttt{in}\, m.\mathbf{0} \mid Q] \mid m[R] \twoheadrightarrow m[n[Q] \mid R]$ if $R \neq \mathbf{0}$	(wRed In 4)
$n[\texttt{in}\, m.P \mid Q] \mid m[\mathbf{0}] \twoheadrightarrow m[n[P \mid Q]]$ if $P \neq \mathbf{0}$	(wRed In 5)
$n[\texttt{in}\, m.P \mid Q] \mid m[R] \twoheadrightarrow m[n[P \mid Q] \mid R]$ if $P \neq \mathbf{0}$ and $R \neq \mathbf{0}$	(wRed In 6)
$m[n[\texttt{out}\, m.P]] \twoheadrightarrow n[P] \mid m[\mathbf{0}]$	(wRed Out 1)
$m[n[\texttt{out}\, m.P] \mid R] \twoheadrightarrow n[P] \mid m[R]$	(wRed Out 2)
$m[n[\texttt{out}\, m.\mathbf{0} \mid Q]] \twoheadrightarrow n[Q] \mid m[\mathbf{0}]$	(wRed Out 3)
$m[n[\texttt{out}\, m.\mathbf{0} \mid Q] \mid R] \twoheadrightarrow n[Q] \mid m[R]$	(wRed Out 4)
$m[n[\texttt{out}\, m.P \mid Q]] \twoheadrightarrow n[P \mid Q] \mid m[\mathbf{0}]$ if $P \neq \mathbf{0}$	(wRed Out 5)
$m[n[\texttt{out}\, m.P \mid Q] \mid R] \twoheadrightarrow n[P \mid Q] \mid m[R]$ if $P \neq \mathbf{0}$	(wRed Out 6)
$!P \twoheadrightarrow P \mid !P$	(wRed Repl 1)
$!P \twoheadrightarrow !P \mid !P$	(wRed Repl 2)
$P \twoheadrightarrow Q \Rightarrow P \mid R \twoheadrightarrow Q \mid R$	(wRed Par)
$P \twoheadrightarrow Q \Rightarrow n[P] \twoheadrightarrow n[Q]$	(wRed Amb)
$P' =_{\mathsf{AC}} P, P \twoheadrightarrow Q, Q =_{\mathsf{AC}} Q' \Rightarrow P' \twoheadrightarrow Q'$	(wRed $=_{\mathsf{AC}}$)

Due to required normalization in presence of $\mathbf{0}$, several rules have been introduced for reductions of the `in` and `out` capabilities; moreover, one rule has been added for the reduction of replication; it aims to simulate the weak reduction $!P \cong !!P \rightarrow_{\mathrm{w}} !!P \mid !P \cong !P \mid !P$. It is easy to see that if P is in normal form and $P \twoheadrightarrow Q$ then Q is in normal form as well. Moreover, we have the following property:

Proposition 3. *For all processes P, Q, let $\tilde{P}, \tilde{Q}$ be their respective normal forms. Then $P \rightarrow_w^* Q$ iff $\tilde{P} \twoheadrightarrow^* \tilde{Q}$.*

Proposition 3 states that the reachability problem for the weak reduction relation can be reduced to a similar problem but restricted to normalized processes. This implies in particular that the use of weak structural congruence has been replaced by the simpler [3] relation of equality modulo associativity and commutativity.

• **From normal processes to Petri nets:** We first show here how to build up a Petri net from a normalized process: roughly speaking, this Petri net aims to encode all the possible reductions over this process. We will show later how to use this Petri net to solve the reachability problem for processes.

Note that applying a reduction over a process either increases the number of ambients in the process or leaves it unchanged: more precisely, when the rule (wRed Repl 1) or (wRed Repl 2) is applied on some process R at some subprocess P containing an ambient then the rule lets the number of ambients increased in the resulting process; other kinds of reduction steps leave the number of ambients unchanged. As we want to decide for two given normalized processes P and Q whether $P \twoheadrightarrow^* Q$, the target process Q is fixed and the number of its ambients is known. Therefore, this can be used to provide an upper-bound on the maximal number of applications of the rules (wRed Repl 1) and (wRed Repl 2) when applied to some subprocess containing an ambient. A similar argument doesn't hold for capabilities as they can be consumed by the reduction rules for the `in` and `out` capabilities.

[3] For equality modulo associativity and commutativity, every congruence class is finite.

This remark leads us to split a process into several parts; intuitively, one of those parts will be a context containing ambients whereas the other ones will be subprocesses without ambients. An *ambient context* C is a process in which may occur some holes, noted as $\Box$. Moreover, we require that in any subcontext $!C'$ of C, C' contains some ambient. Together with ambient contexts, we consider substitutions mapping occurrences of holes to ambient-free processes. Hence, the application of one of these substitutions to an ambient context yields a process.

We will need to refer to a precise occurrence of replication, ambient, capability or hole $\Box$ within an ambient context or a process. Therefore, we are going to introduce a labeling for those objects to be able to distinguish any two of them. We assume for that a countable set of labels. We say that a process P or an ambient context C is *well-labeled* if any label occurs at most once in P or C. For an ambient context C, we define $\mathit{Amb}(C)$ as the multiset of ambients in C.

A labeled transition system: for the reachability problem $S \twoheadrightarrow^* T$, we consider C_S a well-labeled ambient context as well as a mapping θ_S from the set of holes in C_S to labeled ambient-free processes of the form $!P$ such that $\theta_S(C_S)$ is well-labeled and $\theta_S(C_S) = S$ ignoring labels. We are going to describe as a labeled transition system $L_{S,T}$ all possible reductions for the context C_S: this includes reductions of replications and capabilities contained in C_S as well as capabilities and replications from processes associated with the holes of the context.

We consider here a labeled transition system $L_{S,T}$ whose states are AC-equivalent classes of ambient contexts (for simplicity, we often identify a state as one of the representants of its class). We also define a mapping $\theta_{L_{S,T}}$ extending θ_S. Initially, $L_{S,T}$ contains (the equivalence class of) C_S as a unique state and $\theta_{L_{S,T}} = \theta_S$. We iterate the following construction steps until $L_{S,T}$ is unchanged (we assume that any reduction through (wRed In i), (wRed Out i), (wRed Repl 1) and (wRed Repl 2) uses implicitly (wRed Amb), (wRed Par) and (wRed $=_{\mathsf{AC}}$)):

1. for any ambient context C from $L_{S,T}$, for any labeled capability $\mathtt{cap}^w n$ ($\mathtt{cap} \in \{\mathtt{in}, \mathtt{out}\}$) in C if this capability can be executed using one of the rules (wRed In i) or (wRed Out i) leading to some ambient context C', then the state C' and a transition from C to C' labeled by $\mathtt{cap}^w n$ are added to $L_{S,T}$.
2. for any ambient context C from $L_{S,T}$, for any labeled replication $!^w$ in C such that this replication can be reduced using the rule (wRed Repl 1) (resp. (wRed Repl 2)), we define the ambient context C' as follows: C' is identical to C except that the subcontext $!^w C_a$ in C is replaced by $!^w C_a \mid \gamma(C_a)$ (resp. $!^w C_a \mid \gamma(!^w C_a)$) in C'; the mapping γ relabels C_a (resp. $!^w C_a$) with fresh labels, that is labels occurring neither in some other state of the currently built transition system $L_{S,T}$ nor in the currently built $\theta_{L_{S,T}}$; moreover, we require that C' is well-labeled. If $\mathit{Amb}(C') \subseteq \mathit{Amb}(T)$ then the state C' and a transition from C to C' labeled by $!^w_{①}$ (resp. $!^w_{②}$) are added to $L_{S,T}$. Additionally, we define $\theta'_{L_{S,T}}$ as an extension of $\theta_{L_{S,T}}$ such that for all $\Box^{w'}$ in C_a, (i) $\theta'_{L_{S,T}}(\gamma(\Box^{w'}))$ and $\theta_{L_{S,T}}(\Box^{w'})$ are identical ignoring labels, (ii) labels in $\theta'_{L_{S,T}}(\gamma(\Box^{w'}))$ are fresh in the currently built transition system $L_{S,T}$ and in $\theta_{L_{S,T}}$ and (iii) $\theta'_{L_{S,T}}(C')$ is well-labeled. Finally, we set $\theta_{L_{S,T}}$ to $\theta'_{L_{S,T}}$.

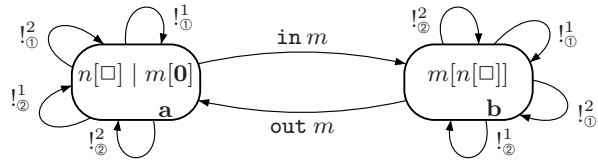


Fig. 1. A labeled transition system for the process $n[!^1\texttt{in}\ m.!^2\texttt{out}\ m.\mathbf{0}] \mid m[\mathbf{0}]$

3. for any ambient context C from $L_{S,T}$, for any labeled hole $\Box^w$ in C and for any capability $\texttt{cap}^{w'}\ n$ (with $\texttt{cap} \in \{\texttt{in}, \texttt{out}\}$) in the process $\theta_{L_{S,T}}(\Box^w)$, we consider the ambient context C_m identical to C except that $\Box^w$ in C has been replaced by $\Box^w \mid \texttt{cap}^{w'} n.\mathbf{0}$ in C_m. If this capability $\texttt{cap}^{w'} n$ can be executed in C_m using one of the rules (wRed In i) or (wRed Out i) leading to some ambient context C', then the state C' and a transition from C to C' labeled by $\texttt{cap}^{w'} n$ are added to $L_{S,T}$.
4. for any ambient context C from $L_{S,T}$, for any labeled hole $\Box^w$ in C associated by $\theta_{L_{S,T}}$ with a process of the form $!^{w'} P$, if the replication $!^{w'}$ can be reduced in the process $\theta_{L_{S,T}}(C)$ using the rule (wRed Repl 1), then for any replication $!^{w''}$ in $\theta_{L_{S,T}}(\Box^w)$, two transitions from C to itself, the first one labeled by $!^{w''}_{①}$ and the second one by $!^{w''}_{②}$ are added to $L_{S,T}$.

It should be noticed that in step 2 the reduction of a replication contained in the ambient context by means of the rule (wRed Repl 1) or (wRed Repl 2) is done only when the number of ambients in the resulting process is smaller than the number of ambients in the target process T. This requirement is crucial as it implies that the transition system $L_{S,T}$ has only finitely many states.

As an example, we give in Figure 1 the labeled transition system associated with the process $n[!^1\texttt{in}\ m.!^2\texttt{out}\ m.\mathbf{0}] \mid m[\mathbf{0}]$ (we omit in this process unnecessary labels).

One can also notice that the labeled transitions in $L_{S,T}$ for replications and capabilities from the ambient context correspond effectively to reductions performed on processes. Things are different for transitions corresponding to replications and capabilities contained in processes associated with holes, as shown in steps 3 and 4: these transitions are applied for any kind of those capabilities or replications independently of the fact that they are effectively at this point available to perform a transition.

We will solve this first by giving a model of processes corresponding to holes as Petri nets and then by synchronizing our two models: the Petri nets and the labeled transition system $L_{S,T}$.

From ambient-free processes to Petri nets: we show here how to build a Petri net from a labeled ambient-free process different from $\mathbf{0}$. For a set E, we denote $\mathcal{E}(E)$ the set of all multisets that can be built with elements from E. We recall that a Petri net is given by a 5-tuple $(\mathcal{P}, \mathcal{P}_i, \mathcal{T}, \mathit{Pre}, \mathit{Post})$ such that $\mathcal{P}$ is a finite set of *places*, $\mathcal{P}_i \subseteq \mathcal{P}$ is a set of *initial places*, $\mathcal{T}$ is a finite set of transitions and $\mathit{Pre}, \mathit{Post} : \mathcal{T} \to \mathcal{E}(\mathcal{P})$ are mappings from transitions to multisets of places. We say that an ambient-free process is *rooted* if it is of the form $\texttt{cap}^w\ n.P$ for $\texttt{cap} \in \{\texttt{in}, \texttt{out}\}$ or of the form $!^w P$. We define PN_P the Petri net associated with some rooted process P as follows: places for PN_P are precisely rooted subprocesses of P, and P itself is the unique initial place. Transitions

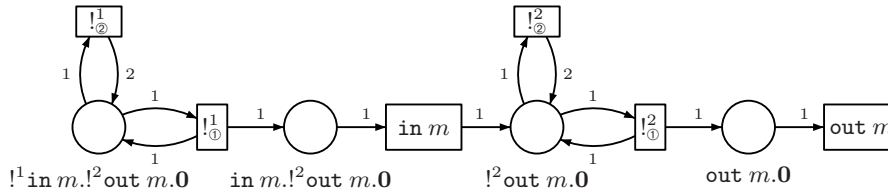


Fig. 2. A Petri net for the process !^{1}in m.!^{2}out m.$\mathbf{0}$

are defined as the set of all capabilities in ^{w} n, out $^{w'}$ n occurring in P and of all $!^w_①$,$!^w_②$ for replications $!^w$ occurring in P. Finally, *Pre* and *Post* are defined for all transitions as follows (for cap $\in \{$in, out$\}$):

- *Pre*(cap^w n) $= \{$cap^w $n.\mathbf{0}\}$ and *Post*(cap^w n) $= \varnothing$ if cap^w $n.\mathbf{0}$ is a place in PN_P.
- *Pre*(cap^w n) $= \{$cap^w $n.(P_1 \mid \ldots \mid P_k)\}$ and *Post*(cap^w n) $= \{P_1, \ldots, P_k\}$ if cap^w $n.(P_1 \mid \ldots \mid P_k)$ is a place in PN_P ($P_1, \ldots, P_k$ being rooted processes).
- *Pre*$(!^w_①) =$ *Pre*$(!^w_②) = \{!^w P\}$, *Post*$(!^w_①) = \{!^w P, P\}$ and *Post*$(!^w_②) = \{!^w P, !^w P\}$ if $!^w P$ is a place in PN_P.

For !^{1}in m.!^{2}out m.$\mathbf{0}$, we obtain the Petri net given in Figure 2.

We will denote $PN_{\Box^w}$ the Petri net $PN(\theta_{L_{S,T}}(\Box^w))$, that is the Petri net corresponding to the rooted ambient-free process associated with $\Box^w$ by $\theta_{L_{S,T}}$.

We will show now how to combine the transition system $L_{S,T}$ and the Petri nets $PN_{\Box^w}$ into one single Petri net.

Combining the transition system and Petri nets: We first turn the labeled transition system $L_{S,T}$ into a Petri net $PN_L = (\mathcal{P}_L, \mathcal{P}^i_L, \mathcal{T}_L, \mathit{Pre}_L, \mathit{Post}_L)$. $\mathcal{P}_L$ is the set of states of $L_{S,T}$. $\mathcal{P}^i_L$ is a singleton set containing the state corresponding to C_S, the ambient context of S. The set of transitions $\mathcal{T}_L$ is the set of triples (s, l, s') where s, s' are states from $L_{S,T}$ with a transition labeled with l from s to s' in $L_{S,T}$. For all transitions $t = (s, l, s')$, *Pre*$(t) = \{s\}$ and *Post*$(t) = \{s'\}$.

We define the Petri net $PN_{S,T} = (\mathcal{P}_{S,T}, \mathcal{P}^i_{S,T}, \mathcal{T}_{S,T}, \mathit{Pre}_{S,T}, \mathit{Post}_{S,T})$ as follows: places (resp. initial places) from $PN_{S,T}$ are the union of places (resp. initial places) of PN_L and of each of the Petri nets $PN_{\Box^w}$ (for $\Box^w$ occurring in one of the states of $L_{S,T}$). Transitions of $PN_{S,T}$ are precisely transitions from PN_L. The mappings *Pre*$_{S,T}$ and *Post*$_{S,T}$ are defined as follows: for all transitions t (t being of the form (a, f, b)), (i) *Pre*$_{S,T}(t) = \{a\}$ and *Post*$_{S,T}(t) = \{b\}$ if f doesn't occur as a transition in any of the $PN_{\Box^w}$'s (for $\Box^w$ occurring in one of the states of $L_{S,T}$) and (ii) *Pre*$_{S,T}(t) = \{a\} \cup$ *Pre*$_{\Box^w}(f)$ and *Post*$_{S,T}(t) = \{b\} \cup$ *Post*$_{\Box^w}(f)$ if f is a transition of $PN_{\Box^w}$ (*Pre*$_{\Box^w}$ (resp. *Post*$_{\Box^w}$) being the mapping *Pre* (resp. *Post*) of $PN_{\Box^w}$).

We depict in Figure 3 the combination of the labeled transition system from Figure 1 and the Petri net from Figure 2.

Deciding reachability: We recall that for a Petri net $PN = (\mathcal{P}, \mathcal{P}_i, \mathcal{T}, \mathit{Pre}, \mathit{Post})$, a marking m is multiset from $\mathcal{E}(\mathcal{P})$. We say that a transition t is enabled by a marking m if *Pre*$(t) \subseteq m$. Firing the enabled transition t for the marking m gives the marking m' defined as $m' = (m \setminus \mathit{Pre}(t)) \cup \mathit{Post}(t)$ (where $\setminus$ stands for the multiset difference). A marking m' is reachable from m if there exists a sequence $m_0, \ldots, m_k$ of markings

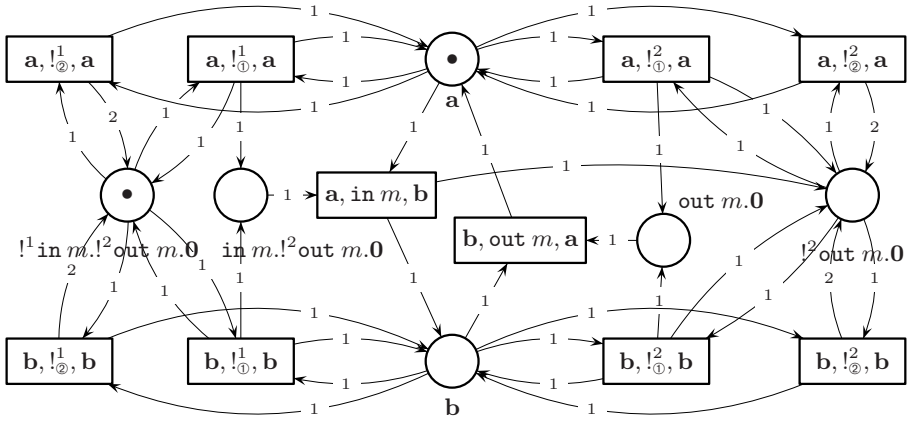


Fig. 3. The Petri net for the labeled process $n[!^1\texttt{in}\ m.!^2\texttt{out}\ m.\mathbf{0}] \mid m[\mathbf{0}]$

such that $m_0 = m$, $m_k = m'$ and for each m_i, m_{i+1}, there exists an enabled transition for m_i whose firing gives m_{i+1}.

Theorem 4. *[15] For all Petri nets P, for all markings m, m' for P, one can decide whether m' is reachable from m.*

For the reachability problem $S \twoheadrightarrow^* T$, we consider the Petri net $PN_{S,T}$ and the initial marking m_S defined as $m_S = \mathcal{P}^i_{S,T}$. In Figure 3 is depicted the initial marking for the process $n[!^1\texttt{in}\ m.!^2\texttt{out}\ m.\mathbf{0}] \mid m[\mathbf{0}]$.

It should be noticed that for any marking m reachable from m_S, m contains exactly one occurrence of a place from $\mathcal{P}_L$. Roughly speaking, to any reachable marking corresponds exactly one ambient context. Moreover, the firing of one transition in the Petri net $PN_{S,T}$ simulates a reduction from $\twoheadrightarrow$. Thus, markings reachable from m_s correspond to normalized processes reachable from S.

We define now $\mathcal{M}_T$, the set of markings of $PN_{S,T}$ corresponding to T. Intuitively, a marking m belongs to $\mathcal{M}_T$ if m contains exactly one occurrence C of a place from $\mathcal{P}_L$ (that is, representing some ambient context) and in the context C, the holes can be replaced with ambient-free processes to obtain T. Moreover, each of those replication-free processes must correspond to a marking of the sub-Petri net associated with the hole it fills up. Formally, $\mathcal{M}_T$ is the set of markings m for $PN_{S,T}$ satisfying: (i) there exists exactly one ambient context C_m in m, (ii) ignoring labels, $\sigma_m(C_m)$ is equal to T modulo AC, for the substitution σ_m from holes $\square^w$ occurring in C_m to ambient-free processes defined as: $\sigma_m(\square^w) = P_1 \mid \ldots \mid P_k$ for $\{P_1, \ldots, P_k\}$ the multiset corresponding to the restriction of m to the places of $PN_{\square^w}$ and (iii) for all holes $\square^w$ occurring in some state of the transition system $L_{S,T}$ but not in C_m, the restriction of m to places of $PN_{\square^w}$ is precisely the set of initial places from $PN_{\square^w}$.

Proposition 4. *For the Petri net $PN_{S,T}$ built from a problem "$S \twoheadrightarrow^* T$", there are only finitely many markings corresponding to T, and their set $\mathcal{M}_T$ can be computed.*

The correctness of our reduction is stated in the following proposition which together with Theorem 4 implies Theorem 3:

Proposition 5. *For all normalized processes S, T, $S \twoheadrightarrow^* T$ iff there exists a marking m_T from $\mathcal{M}_T$ such that m_T is reachable from m_S in $PN_{S,T}$.*

5 On Decision Problems of Name-Convergence and Model-Checking

In this section, we investigate two problems closely related to reachability : the name-convergence problem and the model-checking problem.

5.1 The Name-Convergence Problem

A process P *converges to a name* n if there exists a process P' such that P reduces to P' and P' is structurally congruent to $n[Q] \mid R$ (for some processes Q, R) [10]. The name-convergence problem is, given some process P and some name n, to decide whether P converges to the name n. We are going to show that this problem is not decidable for the two versions of the calculus we have considered so far.

In Section 3, we define the acceptance of an integer v by a two-counters machine $\mathcal{M}$ when $(q_i, v, 0) \vdash^*_{\mathcal{M}} (q_f, 0, 0)$ where q_i, q_f are respectively the initial and the final states of the machine $\mathcal{M}$ and $\vdash^*_{\mathcal{M}}$ is the reflexive-transitive closure of the step relation defined by the machine $\mathcal{M}$. This acceptance condition can be weakened as follows: we say that $\mathcal{M}$ accepts v if there exists two natural numbers v_1, v_2 such that $(q_i, v, 0) \vdash^*_{\mathcal{M}} (q_f, v_1, v_2)$. It is well-known that those two acceptance conditions lead to equally expressive two-counters machines [17].

Reconsidering the encoding given in Section 3, it can be proved that

Proposition 6. *For any two-counters machine $\mathcal{M} = (\mathcal{Q}, q_i, q_f, \Delta)$ and any natural v, the process $[\![(q_i, v, 0)]\!]$ converges to the name q_f iff there exist two natural numbers v_1, v_2, such that $[\![(q_i, v, 0)]\!] \rightarrow^* [\![(q_f, v_1, v_2)]\!]$.*

Now, for the weak calculus, it can be shown that

Proposition 7. *For all naturals v, v_1, v_2, if $[\![(q_i, v, 0)]\!] \rightarrow^* [\![(q_f, v_1, v_2)]\!]$ then there exists a process R such that $[\![(q_i, v, 0)]\!] \rightarrow^*_w [\![(q_f, v_1, v_2)]\!] \mid R$.*

Thus, using the fact that the acceptance of a natural number by an arbitrary two-counters machine is undecidable and Propositions 6 and 7, it holds that

Theorem 5. *The name-convergence problem is undecidable both for the* `in`/`out` *ambient calculus and for the weak* `in`/`out` *ambient calculus.*

5.2 The Model-Checking Problem

The model-checking problem is to decide whether an ambient process satisfies (that is, is a model of) a given formula. Formulas that we consider here are the ones from the ambient logic [4]. The ambient logic is a modal logic used to specify properties of an ambient process; those modalities allow to speak both about time (that is, how a process

can evolve by reduction) and space (that is, what is the shape of the tree description of a process). We will not describe the full logic, but focus on features of our interest.

Any process P satisfies the formula $\mathbf{T}$. A process P satisfies the formula $\Diamond\varphi$ if P can be reduced to some process Q ($P \rightarrow^* Q$ or $P \rightarrow_{\mathrm{w}}^* Q$, depending on the considered calculus) such that Q satisfies the formula φ. A process P satisfies the formula $n[\varphi]$ if P is structurally congruent to some process $n[Q]$ ($P \equiv n[Q]$ or $P \cong n[Q]$, depending on the considered calculus) and Q satisfies φ. Finally, a process P satisfies the formula $\varphi \mid \psi$ if P is congruent to some process $Q \mid R$ and Q, R satisfy respectively φ and ψ.

Proposition 8. *A process P converges to the name n iff P satisfies $\Diamond(n[\mathbf{T}] \mid \mathbf{T})$.*

Using Theorem 5 and Proposition 8, we have

Theorem 6. *The model-checking problem for the* `in`/`out` *ambient calculus and for the weak* `in`/`out` *ambient calculus against the ambient logic is undecidable.*

References

1. T. Amtoft, A. J. Kfoury, and S. M. Pericás-Geertsen. What are polymorphically-typed ambients? In *10th European Symposium on Programming (ESOP 2001)*, LNCS 2028, pages 206–220. Springer, 2001.
2. M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *Theoretical Aspects of Computer Software (TACS 2001)*, LNCS 2215. Springer, 2001.
3. N. Busi and G. Zavattaro. On the expressiveness of movement in pure mobile ambients. In *Foundations of Wide Area Network Computing*, ENTCS 66(3). Elsevier, 2002.
4. L. Cardelli and A.D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *27th Symp. on Principles of Programming Languages (POPL'00)*, pages 365–377, 2000.
5. L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240:177–213, 2000.
6. W. Charatonik, A. D. Gordon, and J.-M. Talbot. Finite-control mobile ambients. In *European Symposium on Programming (ESOP'02)*, LNCS 2305, pages 295–313. Springer, 2002.
7. W. Charatonik and J.-M. Talbot. The decidability of model checking mobile ambients. In *Computer Science Logic (CSL'01)*, LNCS 2142, pages 339–354. Springer, 2001.
8. S. Dal Zilio. Spatial congruence for ambients is decidable. In *6th Asian Computing Science Conference (ASIAN'00)*, volume 1961 of *LNCS*, pages 88–103. Springer, 2000.
9. J. Feret. Abstract interpretation-based static analysis of mobile ambients. In *Eighth International Static Analysis Symposium (SAS'01)*, LNCS 2126. Springer, 2001.
10. A. D. Gordon and L. Cardelli. Equational properties of mobile ambients. *Mathematical Structures in Computer Science*, 12:1–38, 2002.
11. R.R. Hansen, J.G. Jensen, F. Nielson, and H. Riis Nielson. Abstract interpretation of mobile ambients. In *Static Analysis (SAS'99)*, LNCS 1694, pages 134–148. Springer, 1999.
12. D. Hirschkoff. *Mise en œuvre de preuves de bisimulation*. PhD thesis, Ecole Nationale des Ponts et Chaussées, 1999.
13. D. Hirschkoff, E. Lozes, and D. Sangiorgi. Separability, expressiveness, and decidability in the ambient logic. In *Logic in Computer Science (LICS'02)*, pages 423–432. IEEE, 2002.
14. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *27th Symp. on Principles of Programming Languages (POPL'00)*, pages 352–364, 2000.
15. E.W. Mayr. An Algorithm for the General Petri Net Reachability Problem. *SIAM Journal of Computing*, 13(3):441–460, 1984.

16. R. Milner, J. Parrow, and J. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, 1992.
17. M. Minsky. Recursive Unsolvability of Post's Problem of "Tag" and others Topics in the Theory of Turing Machines. *Annals of Math.*, 74:437–455, 1961.
18. F. Nielson, H. Riis Nielson, R.R. Hansen, and J.G. Jensen. Validating firewalls in mobile ambients. In *Concurrency Theory (Concur'99)*, LNCS 1664, pages 463–477. Springer, 1999.
19. D. Teller, P. Zimmer, and D. Hirschkoff. Using ambients to control resources. In *CONCUR 2002—Concurrency Theory*, LNCS 2421, pages 288–303. Springer, 2002.
20. P. Zimmer. On the Expressiveness of Pure Safe Ambients. *Mathematical Structures of Computer Science*, 2002. To appear.

Computability over an Arbitrary Structure. Sequential and Parallel Polynomial Time

Olivier Bournez[1], Felipe Cucker[2], Paulin Jacobé de Naurois[1⋆], and Jean-Yves Marion[1]

[1] LORIA,
615 rue du Jardin Botanique, BP 101,
54602 Villers-lès-Nancy Cedex, Nancy, France.
{Olivier.Bournez,Paulin.De-Naurois,Jean-Yves.Marion}@loria.fr
[2] City University of Hong Kong,
Tat Chee avenue,
Kowloon, Hong Kong.
macucker@math.cityu.edu.hk

Abstract. We provide several machine-independent characterizations of deterministic complexity classes in the model of computation proposed by L. Blum, M. Shub and S. Smale. We provide a characterization of partial recursive functions over any arbitrary structure. We show that polynomial time computable functions over any arbitrary structure can be characterized in term of safe recursive functions. We show that polynomial parallel time decision problems over any arbitrary structure can be characterized in terms of safe recursive functions with substitutions.

1 Introduction

Why are we convinced by the Church Thesis? An answer is certainly that there are so many mathematical models, like partial recursive functions, lambda-calculus, or semi-Thue systems, which are equivalent to Turing machine, but are also independent from the computational machinery. When computing over arbitrary structures, like real numbers, the situation is not so clear. Seeking machine independent characterizations of complexity classes can lend further credence to the importance of the classes and models considered.

We consider here the BSS model of computation over the real numbers introduced by Blum, Shub and Smale in their seminal paper [BSS89]. The model was later on extended to a computational model over any arbitrary logical structure [Goo94, Poi95]. See the monograph [BCSS98] for a general survey about the BSS model.

First of all, we present a new characterization of computable functions that extends the one of [BSS89] to any arbitrary structure.

⋆ This author has been partially supported by City University of Hong Kong SRG grant 7001290.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 185–199, 2003.

Theorem 1. *Over any structure* $\mathcal{K} = (\mathbb{K}, op_1, \ldots, op_k, =, rel_1, \ldots, rel_l, \alpha)$*, the set of partial recursive functions over* $\mathcal{K}$ *is exactly the set of decision functions computed by a BSS machine over* $\mathcal{K}$*.*

In the BSS model, complexity classes like PTIME and NPTIME can be defined, and complete problems in these classes can be shown to exist. On many aspects, this is an extension of the classical complexity theory since complexity classes correspond to classical complexity classes when dealing with booleans or integers. The next results strengthen this idea.

In classical complexity theory, several attempts have been done to provide nice formalisms to characterize complexity classes in a machine independent way. Such characterizations include descriptive characterization based on finite model theory like Fagin [Fag74], characterization by function algebra like [Cob62], or by combining both kinds of characterization like in [Gur83, Saz80]: see [Clo95, Imm99, EF95] for more complete references.

Despite the success of those approaches to capture major complexity classes, one may not be completely satisfied because explicit upper bounds on computational resources or restrictions on the growth rates are present. The recent works of Bellantoni and Cook [BC92] and of Leivant [Lei95, LM93] suggests another direction by mean of data tiering which is called implicit computational complexity. There are no more explicit resource bounds. It provides purely syntactic models of complexity classes which can be applied to programming languages to analyze program complexity [Jon00, Hof99, MM00].

In this paper, following these lines, we establish two "implicit" characterizations of the complexity classes. Our characterizations work over arbitrary structures, and subsume previous ones when restricted to booleans or integers.

First, we characterize polynomial time computable BSS functions. This result stems on the safe primitive recursion principle of Bellantoni and Cook [BC92].

Theorem 2. *Over any structure* $\mathcal{K} = (\mathbb{K}, op_1, \ldots, op_k, =, rel_1, \ldots, rel_l, \alpha)$*, the set of safe recursive functions over* $\mathcal{K}$ *is exactly the set of functions computed in polynomial time by a BSS machine over* $\mathcal{K}$*.*

Second, we capture parallel polynomial time BSS functions based on Leivant and Marion characterization of polynomial space computable functions [LM95].

Theorem 3. *Over any structure* $\mathcal{K} = (\mathbb{K}, op_1, \ldots, op_k, =, rel_1, \ldots, rel_l, \alpha)$*, the set of decision functions definable with safe recursion with substitution over* $\mathcal{K}$ *is exactly the set of decision functions computed in parallel polynomial time over* $\mathcal{K}$*.*

Observe that, unlike Leivant and Marion, our proofs characterize parallel polynomial time and not polynomial space: for classical complexity both classes correspond. However over arbitrary structures, this is not true, since the notion of working space is meaningless: as pointed out by Michaux [Mic89], on some structures like $(\mathbb{R}, 0, 1, =, +, -, *)$, any computable function can be computed in constant working space.

From a programming perspective, a way of understanding all these results is to see computability over arbitrary structures like a programming language with extra operators which come from some external libraries. This observation, and its potential to able to build methods to derive automatically computational properties of programs, in the lines of [Jon00, Hof99, MM00], is one of our main motivations on making this research.

On the other hand, we believe BSS computational model to provide new insights for understanding complexity theory when dealing with structures over other domains [BCSS98]: several nice results have been obtained for this model in the last decade, including separation of complexity classes over specific structures: see for example [Mee92, Cuc92, CSS94]. We believe these results to contribute to the understanding of complexity theory, even when restricted to classical complexity [BCSS98].

It is worth mentioning that it is not the first time that the implicit computational complexity community is interested by computations over real numbers: see the exciting paper of Cook [Coo92] on higher order functionals, or the works trying to clarify the situation such as [RIR01]. However this is the first time that implicit characterizations of this type over arbitrary structures are given.

In Section 2, we give a characterization of primitive recursive functions over an arbitrary structure. We introduce our notion of safe recursive function in Section 3. We give the proof of Theorem 2 in Section 4. We recall the notion of a family of circuits over an arbitrary structure in Section 5. Safe recursion with substitutions is defined in Section 5.2. Theorem 3 is proved in Section 5.3 and 5.4.

2 Partial Recursive and Primitive Recursive Functions

2.1 Definitions

A structure $\mathcal{K} = (\mathbb{K}, op_1, \ldots, op_k, rel_1, \ldots, rel_l, \alpha)$ is given by some underlying set $\mathbb{K}$, some operators $op_1, \ldots, op_k$ with arities, and some relations $rel_1, \ldots, rel_l$ with arities. Constants correspond to operators of arity 0. We will not distinguish between operator and relation symbols and their corresponding interpretations as functions and relations respectively over the underlying set $\mathbb{K}$.

We assume that equality relation $=$ is always one relation of the structure, and that there is at least one constant α in the structure. A good example for such a structure, corresponding to the original paper in [BSS89] is $\mathcal{K} = (\mathbb{R}, +, -, *, =, \leq, 0, 1)$. Another one, corresponding to classical complexity and computability theory, is $\mathcal{K} = (\{0,1\}, \vee, \wedge, =, 0, 1)$.

$\mathbb{K}^* = \bigcup_{i \in \mathbb{N}} \mathbb{K}^i$ will denote the set of words over alphabet $\mathbb{K}$. In our notations, words of elements in $\mathbb{K}$ will be represented with overlined letters, while simple elements in $\mathbb{K}$ will be represented by simple letters. For instance, $a.\overline{x}$ stands for the word in $\mathbb{K}^*$ whose first letter is a and which ends with the word $\overline{x}$. ϵ will denote the empty word. The length of a word $\overline{w} \in \mathbb{K}^*$ is denoted by $|\overline{w}|$.

We assume that the reader has some familiarities with the computation model of Blum Shub and Smale: see [BSS89, BCSS98] for a detailed presentation.

In particular remember that a problem $P \subset \mathbb{K}^*$ is decidable (respectively a function $f : \mathbb{K}^* \to \mathbb{K}^*$ is computable), if there exists a machine M over structure $\mathcal{K}$ that decides P (resp. computes f). A problem $P \subset \mathbb{K}^*$ is in the class PTIME (respectively a function $f : \mathbb{K}^* \to \mathbb{K}^*$ is in the class FPTIME), if there exists a polynomial p and a machine M over structure $\mathcal{K}$ that decides P (resp. computes f) in time p.

As for the classical settings, computable functions over any arbitrary structure $\mathcal{K}$ can be characterized algebraically, in terms of the smallest set of functions containing some initial functions and closed by composition, primitive recursion and minimization. In the rest of this section we present such a characterization that works over any arbitrary structure. See comments below for comparisons with the one, for the structure of real numbers, in the original paper [BSS89].

We consider functions: $(\mathbb{K}^*)^n \to \mathbb{K}^*$, taking as inputs arrays of words of elements in $\mathbb{K}$, and returning as output a word of elements in $\mathbb{K}$. When the output of a function is undefined, we use the symbol $\perp$.

Theorem 1. *Over any structure $\mathcal{K} = (\mathbb{K}, op_1, \ldots, op_k, =, rel_1, \ldots, rel_l, \alpha)$, the set of functions $(\mathbb{K}^*)^n \to \mathbb{K}^*$ computed by a BSS machine over $\mathcal{K}$ is exactly the set of partial recursive functions, that is to say the smallest set of functions containing the basic functions, and closed under the operations of composition, primitive recursion, and minimization defined below.*

The basic functions are of four kinds:

– functions making elementary manipulations of words of elements in $\mathbb{K}$. For any $a \in \mathbb{K}, \overline{x}, \overline{x_1}, \overline{x_2} \in \mathbb{K}^*$:

$$\begin{array}{rrr} \mathrm{hd}(a.\overline{x}) = a & \mathrm{tl}(a.\overline{x}) = \overline{x} & \mathrm{cons}(a.\overline{x_1}, \overline{x_2}) = a.\overline{x_2} \\ \mathrm{hd}(\epsilon) = \epsilon & \mathrm{tl}(\epsilon) = \epsilon & \mathrm{cons}(\epsilon, \overline{x_2}) = \overline{x_2} \end{array}$$

– Projections: for any $n \in \mathbb{N}$, $\imath \leq n$:

$$\mathrm{Pr}_\imath^n(\overline{x_1}, \ldots, \overline{x_\imath}, \ldots, \overline{x_n}) = \overline{x_\imath}$$

– functions of structure $\mathcal{K}$: for any operator (including the constants treated as operators of arity 0) $op_\imath$ or relation $rel_\imath$ of arity $n_\imath$ we have the following initial functions:

$$\mathrm{Op}_\imath(a_1.\overline{x_1}, \ldots, a_{n_\imath}.\overline{x_{n_\imath}}) = (op_\imath(a_1, \ldots, a_{n_\imath})).\overline{x_{n_\imath}}$$

$$\mathrm{Rel}_\imath(a_1.\overline{x_1}, \ldots, a_{n_\imath}.\overline{x_{n_\imath}}) = \begin{cases} \alpha \text{ if } rel_\imath(a_1, \ldots, a_{n_\imath}) \\ \epsilon \text{ otherwise} \end{cases}$$

– test function :

$$\mathrm{C}(\overline{x}, \overline{y}, \overline{z}) = \begin{cases} \overline{y} & \text{if } \mathrm{hd}(\overline{x}) = \alpha \\ \overline{z} & \text{otherwise} \end{cases}$$

Operations mentioned above are:

- composition: Assume f: $(\mathbb{K}^*)^n \to \mathbb{K}^*$, $g_1, \ldots, g_n$: $(\mathbb{K}^*)^m \to \mathbb{K}^*$ are given partial functions. Then the composition h: $(\mathbb{K}^*)^m \to \mathbb{K}^*$ is defined by

$$h(\overline{x_1}, \ldots, \overline{x_m}) = f(g_1(\overline{x_1}, \ldots, \overline{x_m}), \ldots, g_n(\overline{x_1}, \ldots, \overline{x_m}))$$

- primitive recursion: Assume h: $\mathbb{K}^* \to \mathbb{K}^*$ and g: $(\mathbb{K}^*)^3 \to \mathbb{K}^*$ are given partial functions. Then we define f: $(\mathbb{K}^*)^2 \to \mathbb{K}^*$

$$f(\epsilon, \overline{x}) = h(\overline{x})$$
$$f(a.\overline{y}, \overline{x}) = \begin{cases} g(\overline{y}, f(\overline{y}, \overline{x}), \overline{x}) \text{ if } f(\overline{y}, \overline{x}) \neq \perp \\ \perp \qquad\qquad\qquad\quad \text{otherwise} \end{cases}$$

- minimization: Assume $f : (\mathbb{K}^*)^2 \to \mathbb{K}^*$ is given. Function g: $\mathbb{K}^* \to \mathbb{K}^*$ is define by minimization on the first argument of f, also denoted by $g(\overline{y}) = \mu\overline{x}\,(f(\overline{x}, \overline{y}))$, if:

$$\mu\overline{x}\,(f(\overline{x}, \overline{y})) = \begin{cases} \perp & \text{if } \forall t \in \mathbb{N} \ : \mathrm{hd}(f(0^t, \overline{y})) \neq \alpha \\ \alpha^k : \ k = \min\{t \mid \mathrm{hd}(f(0^t, \overline{y})) = \alpha\} & \text{otherwise} \end{cases}$$

Note 1. Our definition of primitive recursion and of minimization is sightly different from the one found in [BSS89]. In this paper, the authors introduce a special integer argument for every function, which is used to control recursion and minimization, and consider the other arguments as simple elements in $\mathbb{K}$. Their functions are of type: $\mathbb{N} * \mathbb{K}^k \to \mathbb{K}^l$. Therefore, they only capture finite dimensional functions. It is known that, on the real numbers with $+, -, *$ operators, finite dimensional functions are equivalent to non-finite dimensional functions (see [Mic89]), but this is not true over other structures, for instance $\mathbb{Z}_2$. Our choice is to consider arguments as words of elements in $\mathbb{K}$, and to use the length of the arguments to control recursion and minimization. This allows us to capture non-finite dimensional functions,. We consider it to be a more general and a more natural way to define computable functions, and moreover not restricted to structure $\mathcal{K} = (\mathbb{R}, +, -, *, =, \leq, 0, 1)$.

Observe that primitive recursion can be replaced by simultaneous primitive recursion:

Proposition 1. [BMdN02] *Simultaneous primitive recursion is definable with primitive recursive functions.*

The proof of Theorem 1, similar to the proof of Theorem 2 in section 3, is not given here.

3 Safe Recursive Functions

In this section we define the set of safe recursive functions over any arbitrary structure $\mathcal{K}$, extending the notion of safe recursive functions over the natural numbers found in [BC92].

Safe recursive functions are defined in a quite similar manner as primitive recursive functions. However, in the spirit of [BC92], safe recursive functions have two different types of arguments, each of which having different properties and different purposes. The first type of argument, called "normal" arguments, is similar to the arguments of the previously defined partial recursive and primitive recursive functions, since it can be used to make basic computation steps or to control recursion. The second type of argument is called "safe", and can not be used to control recursion. This distinction between safe and normal arguments ensures that safe recursive functions can be computed in polynomial time.

We will use the following notations: the two different types of arguments are separated by a semicolon ";" : normal arguments (respectively: safe arguments) are placed at left (resp. at right) of the semicolon.

We define now safe recursive functions:

Definition 1. *The set of safe recursive functions over* $\mathbb{K}$ *is the smallest set of functions:* $(\mathbb{K}^*)^k \to \mathbb{K}^*$*, containing the basic safe functions, and closed under the operations of safe composition and safe recursion*

Basic safe functions are the basic functions of Section 2, their arguments defined all as safe.

Operations mentioned above are:

- safe composition: Assume f: $(\mathbb{K}^*)^m \times (\mathbb{K}^*)^n \to \mathbb{K}^*$, $g_1, g_m : (\mathbb{K}^*)^p \to \mathbb{K}^*$ and $g_{m+1}, g_{m+n} : (\mathbb{K}^*)^p \times (\mathbb{K}^*)^q \to \mathbb{K}^*$ are given functions. Then the composition is the function $h : (\mathbb{K}^*)^p \times (\mathbb{K}^*)^q \to \mathbb{K}^*$:

$$\begin{aligned} h(\overline{x_1}, \ldots, \overline{x_p}; \overline{y_1}, \ldots, \overline{y_q}) = f\,(&g_1(\overline{x_1}, \ldots, \overline{x_p}), \ldots, g_m(\overline{x_1}, \ldots, \overline{x_p}); \\ &g_{m+1}(\overline{x_1}, \ldots, \overline{x_p}; \overline{y_1}, \ldots, \overline{y_q}), \ldots, g_{m+1}(\overline{x_1}, \ldots, \overline{x_p}; \overline{y_1}, \ldots, \overline{y_q})) \end{aligned}$$

  Note 2. It is possible to move an argument from the normal position to the safe position, whereas the reverse is forbidden. By "move", we mean the following: for example, assume $g : \mathbb{K}^* \times (\mathbb{K}^*)^2 \to \mathbb{K}^*$ is a given function. One can then define with safe composition a function f: $f(\overline{x}, \overline{y}; \overline{z}) = g(\overline{x}; \overline{y}, \overline{z})$ but a definition like the following is not valid: $f(\overline{x}; \overline{y}, \overline{z}) = g(\overline{x}, \overline{y}; \overline{z})$.

- safe recursion: Assume $h_1, \ldots, h_k : \mathbb{K}^* \times \mathbb{K}^* \to \mathbb{K}^*$ and $g_1, \ldots, g_k : (\mathbb{K}^*)^2 \times (\mathbb{K}^*)^{k+1} \to \mathbb{K}^*$ are given functions. Functions $f_1, \ldots, f_k : (\mathbb{K}^*)^2 \times \mathbb{K}^* \to \mathbb{K}^*$ can then be defined by safe recursion:

$$f_1(\epsilon, \overline{x}; \overline{y}), \ldots, f_k(\epsilon, \overline{x}; \overline{y}) = h_1(\overline{x}; \overline{y}), \ldots, h_k(\overline{x}; \overline{y})$$

$$f_1(a.\overline{z}, \overline{x}; \overline{y}) = \begin{cases} g_1(\overline{z}, \overline{x}; f_1(\overline{z}, \overline{x}; \overline{y}), \ldots, f_k(\overline{z}, \overline{x}; \overline{y}), \overline{y}) & \text{if } \forall \imath\ f_\imath(\overline{z}, \overline{x}; \overline{y}) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$\vdots$$

$$f_k(a.\overline{z}, \overline{x}; \overline{y}) = \begin{cases} g_k(\overline{z}, \overline{x}; f_1(\overline{z}, \overline{x}; \overline{y}), \ldots, f_k(\overline{z}, \overline{x}; \overline{y}), \overline{y}) & \text{if } \forall \imath\ f_\imath(\overline{z}, \overline{x}; \overline{y}) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Note 3. The operation of primitive recursion previously defined is a simple recursion, whereas the operation of safe recursion is a simultaneous recursion. As stated by Proposition 1, it is possible to simulate a simultaneous primitive recursion with single primitive recursion, whereas this does not seem to be true with safe recursion. As shown in the simulation of a BSS machine by safe recursive functions, we need to have a simultaneous recursion able to define simultaneously three functions in order to prove Theorem 2. In the classical setting, this is the choice made by Leivant and Marion in [LM95], while Bellantoni and cook used a smash function # to build and break t-uples [BC92]. Both choices are equivalent.

4 Proof of Theorem 2

Theorem 2 is proved in two steps. First, we prove that a safe recursive function can be computed by a BSS machine in polynomial time. Second, we prove that all functions computable in polynomial time by a BSS machine over $\mathbb{K}$ can be expressed as safe recursive functions.

4.1 Polynomial Time Evaluation of a Safe Recursive Function

This is a straightforward consequence of the following lemma.

Lemma 1. *Let $f(\overline{x_1},\ldots,\overline{x_n};\overline{y_1},\ldots,\overline{y_m})$ be any safe recursive function. If we write $T^{\ulcorner} f(\ldots)^{\urcorner}$ for the evaluation time of $f(\ldots)$,*

$$T^{\ulcorner} f(\overline{x_1},\ldots,\overline{x_n};\overline{y_1},\ldots,\overline{y_m})^{\urcorner} \leq p_f(T^{\ulcorner}\overline{x_1}^{\urcorner}+\ldots+T^{\ulcorner}\overline{x_n}^{\urcorner})+T^{\ulcorner}\overline{y_1}^{\urcorner}+\ldots+T^{\ulcorner}\overline{y_m}^{\urcorner}$$

for some polynomial p_f.

This is proved by induction on the depth of the definition tree of the safe recursive function. Let f be a safe recursive function.

- If f is a basic safe function, the result is straightforward.
- if f is defined by safe composition from safe recursive functions g, h_1, h_2, using induction hypothesis, f is easily shown to be computable in polynomial time by a BSS machine.
- The non-trivial case is the case of a function f defined with simultaneous safe recursion. In order to simplify the notations, we assume that f is defined with a single safe recursion. The proof is in essence the same.
  Let us apply induction hypothesis to function g in the expression $f(a.\overline{z},\overline{x};\overline{y}) = g(\overline{z},\overline{x};f(\overline{z},\overline{x};\overline{y}),\overline{y})$. Assuming a "clever" strategy, $\overline{y}$ needs to be evaluated only once, even though it appears in several recursive calls. Thus, if we assume that $\overline{y}$ has already been evaluated, the time needed to evaluate $f(a.\overline{z},\overline{x};\overline{y}) = g(\overline{z},\overline{x};f(\overline{z},\overline{x};\overline{y}),\overline{y})$ is given by $p_g(T^{\ulcorner}\overline{z}^{\urcorner}+T^{\ulcorner}\overline{x}^{\urcorner})+T^{\ulcorner}f(\overline{z},\overline{x};\overline{y})^{\urcorner}$.

We get:

$$\begin{aligned} & T^{\ulcorner} f(a.\overline{z}, \overline{x}; \overline{y})^{\urcorner} \\ \leq\ & T^{\ulcorner}\overline{y}^{\urcorner} + p_g(T^{\ulcorner}\overline{z}^{\urcorner} + T^{\ulcorner}\overline{x}^{\urcorner}) + T^{\ulcorner} f(\overline{z}, \overline{x}; \overline{y})^{\urcorner} \\ \leq\ & T^{\ulcorner}\overline{y}^{\urcorner} + p_g(T^{\ulcorner}\overline{z}^{\urcorner} + T^{\ulcorner}\overline{x}^{\urcorner}) + p_g(T^{\ulcorner}\overline{\mathrm{tl}(z)}^{\urcorner} + T^{\ulcorner}\overline{x}^{\urcorner}) + \ldots \\ & \qquad \ldots + p_g(T^{\ulcorner}\overline{\epsilon}^{\urcorner} + T^{\ulcorner}\overline{x}^{\urcorner}) + p_h(T^{\ulcorner}\overline{x}^{\urcorner}) \,. \end{aligned}$$

Assuming without loss of generality p_g monotone, we get

$$T^{\ulcorner} f(a.\overline{z}, \overline{x}; \overline{y})^{\urcorner} \leq |a.\overline{z}| p_g(T^{\ulcorner}\overline{z}^{\urcorner} + T^{\ulcorner}\overline{x}^{\urcorner}) + p_h(T^{\ulcorner}\overline{x}^{\urcorner}) + T^{\ulcorner}\overline{y}^{\urcorner},$$

from which the lemma follows.

4.2 Simulation of a Polynomial Time BSS Machine

Let M be a BSS machine over the structure $\mathcal{K}$. In order to simplify our exposition we assume, without any loss of generality, that M has a single tape. M computes a partial function f_M from $\mathbb{K}^*$ to $\mathbb{K}^*$. Moreover, we assume that M stops after $c|\overline{x}|^r$ computation steps, where $\overline{x}$ denotes the input of the machine M. Our goal is to prove that f_M can be defined as a safe recursive function.

In what follows, we represent the tape of the machine M by a couple a variables $(\overline{y_1}, \overline{y_2})$ in $(\mathbb{K}^*)^2$ such that the non-empty part is given by $\overline{y_1}^R.\overline{y_2}$, where $\overline{y_1}^R$ is the reversed word of $\overline{y_1}$, and that the head of the machine is on the first letter of $\overline{y_2}$.

We also assume that the m nodes in M are numbered with natural numbers, node 0 being the initial node and node 1 the terminal node. We assume that the terminal node is a loopback node, i.e., its only next node is itself. In the following definitions, node number q will be coded by the word α^q of length q.

Let q ($q \in \mathbb{N}$) be a move node. Three functions are associated with this node:

$$\begin{array}{lll} \mathcal{G}_i(;\overline{y_1}, \overline{y_2}) & = \alpha^{q'} & \\ \mathcal{H}_i(;\overline{y_1}, \overline{y_2}) & = \mathrm{tl}(;\overline{y_1}) & \text{or } \mathrm{hd}(;\overline{y_2}).\overline{y_1} \\ \mathcal{I}_i(;\overline{y_1}, \overline{y_2}) & = \mathrm{hd}(;\overline{y_1}).\overline{y_2} & \text{or } \mathrm{tl}(;\overline{y_2}) \end{array}$$

according if one moves right or left. Function $\mathcal{G}_i$ returns the encoding of the following node in the computation tree of M, function $\mathcal{H}_i$ returns the encoding of the left part of the tape, and function $\mathcal{I}_i$ returns the encoding of the right part of the tape.

Let q ($q \in \mathbb{N}$) be a node associated to some operation *op* of arity n of the structure. We also write Op for the corresponding basic operation. Functions $\mathcal{G}_i$, $\mathcal{H}_i$, $\mathcal{I}_i$ associated with this node are now defined as follows:

$$\begin{array}{ll} \mathcal{G}_i(;\overline{y_1}, \overline{y_2}) & = \alpha^{q'} \\ \mathcal{H}_i(;\overline{y_1}, \overline{y_2}) & = \overline{y_1} \\ \mathcal{I}_i(;\overline{y_1}, \overline{y_2}) & = \mathrm{cons}(;\mathrm{Op}(;\mathrm{hd}(;\overline{y_2}), \ldots, \mathrm{hd}(;\mathrm{tl}^{(n-1)}(;\overline{y_2}))), \mathrm{tl}^{(n)}(;\overline{y_2})) \end{array}$$

Let q ($q \in \mathbb{N}$) be a node corresponding to a relation rel of arity n of the structure. The three functions associated with this node are now:

$$\begin{aligned}
\mathcal{G}_i(;\overline{y_1},\overline{y_2}) &= \mathrm{C}(;rel(;\mathrm{hd}(;\overline{y_1}),\ldots,\mathrm{hd}(;\mathrm{tl}^{(n-1)}(;\overline{y_2}))),\alpha^{q'},\alpha^{q''})\\
\mathcal{H}_i(;\overline{y_1},\overline{y_2}) &= \overline{y_1}\\
\mathcal{I}_i(;\overline{y_1},\overline{y_2}) &= \overline{y_2}
\end{aligned}$$

One can define easily without safe recursion, for any integer k, a function $Equal_k$ such that:

$$Equal_k(;\overline{y_1}) = \begin{cases} \alpha \text{ if } \overline{y_1} = \alpha^k \\ \epsilon \text{ otherwise} \end{cases}$$

We can now define the safe recursive functions $next_{state}$, $next_{left}$ and $next_{right}$ which, given the encoding of a state and of the tape of the machine, return the encoding of the next state in the computation tree, the encoding of the left part of the tape and the encoding of the right part of the tape:

$$\begin{aligned}
next_{state}(;\overline{s},\overline{y_1},\overline{y_2}) &= \mathrm{C}\,(;Equal_0(;\overline{s}),\mathcal{G}_0(;\overline{y_1},\overline{y_2}),\mathrm{C}\,(;Equal_1(;\overline{s}),\mathcal{G}_1(;\overline{y_1},\overline{y_2}),\\
&\quad \ldots\mathrm{C}\,(;Equal_m(;\overline{s}),\mathcal{G}_m(;\overline{y_1},\overline{y_2}),\epsilon)\ldots))\\
next_{left}(;\overline{s},\overline{y_1},\overline{y_2}) &= \mathrm{C}\,(;Equal_0(;\overline{s}),\mathcal{H}_0(;\overline{y_1},\overline{y_2}),\mathrm{C}\,(;Equal_1(\overline{s}),\mathcal{H}_1(;\overline{y_1},\overline{y_2}),\\
&\quad \ldots\mathrm{C}\,(;Equal_m(;\overline{s}),\mathcal{H}_m(;\overline{y_1},\overline{y_2}),\epsilon)\ldots))\\
next_{right}(;\overline{s},\overline{y_1},\overline{y_2}) &= \mathrm{C}\,(;Equal_0(;\overline{s}),\mathcal{I}_0(;\overline{y_1},\overline{y_2}),\mathrm{C}\,(;Equal_1(;\overline{s}),\mathcal{I}_1(;\overline{y_1},\overline{y_2}),\\
&\quad \ldots\mathrm{C}\,(;Equal_m(;\overline{s}),\mathcal{I}_m(;\overline{s},\overline{y_1},\overline{y_2}),\epsilon)\ldots))
\end{aligned}$$

From now on, we define with safe recursion the encoding of the state of the machine reached after k computation nodes, where k is encoded by the word $\alpha^k \in \mathbb{K}^*$, and we also define the encoding of the left part and the right part of the tape. All this is done with functions $comp_{state}$, $comp_{left}$ and $comp_{right}$ as follows:

$$\begin{aligned}
comp_{state}(\epsilon;\overline{y_1},\overline{y_2}) &= \epsilon\\
comp_{state}(\alpha^{k+1};\overline{y_1},\overline{y_2}) &= next_{state}\big(;comp_{state}(\alpha^k;\overline{y_1},\overline{y_2}),comp_{left}(\alpha^k;\overline{y_1},\overline{y_2}),\\
&\qquad comp_{right}(\alpha^k;\overline{y_1},\overline{y_2})\big)\\
comp_{left}(\epsilon;\overline{y_1},\overline{y_2}) &= \overline{y_1}\\
comp_{left}(\alpha^{k+1};\overline{y_1},\overline{y_2}) &= next_{left}\big(;comp_{state}(\alpha^k;\overline{y_1},\overline{y_2}),comp_{left}(\alpha^k;\overline{y_1},\overline{y_2}),\\
&\qquad comp_{right}(\alpha^k;\overline{y_1},\overline{y_2})\big)\\
comp_{right}(\epsilon;\overline{y_1},\overline{y_2}) &= \overline{y_2}\\
comp_{right}(\alpha^{k+1};\overline{y_1},\overline{y_2}) &= next_{right}\big(;comp_{state}(\alpha^k;\overline{y_1},\overline{y_2}),comp_{left}(\alpha^k;\overline{y_1},\overline{y_2}),\\
&\qquad comp_{right}(\alpha^k;\overline{y_1},\overline{y_2})\big)
\end{aligned}$$

In order to simplify the notations, we write the above as follows:

$$\begin{aligned}
comp(\epsilon;\overline{y_1},\overline{y_2}) &= \epsilon\\
comp(\alpha^{k+1};\overline{y_1},\overline{y_2}) &= next\big(;comp(\alpha^k;\overline{y_1},\overline{y_2})\big)
\end{aligned}$$

On input $\overline{x}$, with the head originally on the first letter of $\overline{x}$, the final state of the computation of M is then reached after t computation steps, where

$$t = c|\overline{x}|^r$$

The reachability of this final state is given by the following lemma:

Lemma 2. [BMdN02] *For any $c, d \in \mathbb{N}$, one can write a safe recursive function $P_{c,d}$ such that, on any input $\overline{x}$ with $|\overline{x}| = n$, $|P_{c,d}(\overline{x};)| = cn^d$.*

Let us apply this lemma and define such a $P_{c,r}$. Then, we define $finalcomp(\overline{x};) = comp(P_{c,r}(\overline{x};); \epsilon, \overline{x})$. The encoding of the tape at the end of the computation is then given by $finalcomp_{left}(\overline{x};)$ and $finalcomp_{right}(\overline{x};)$ ending here our simulation of the BSS machine M.

5 A Characterization of the Parallel Class $\mathrm{PAR}_{\mathcal{K}}$

5.1 A Parallel Model of Computation

In this section, we assume that our structure $\mathcal{K}$ has at least two different constant, denoted by α and β. This is necessary to respect the P-uniformity provisio as below: One needs to describe an exponential gate number with a polynomially long codification.

Recall the notion of circuit over an arbitrary structure $\mathcal{K}$ [Poi95, BCSS98].

Definition 2. *A circuit over the structure $\mathcal{K}$ is an acyclic directed graph whose nodes are labeled either as input nodes of in-degree 0, output nodes of out-degree 0, test nodes of in-degree 3, or by a relation or an operation of the structure, of in-degree equal to its arity.*

The evaluation of a circuit on a given valuation of input nodes is defined in the straightforward way, all nodes behaving as one would expect: any test node tests whether its first parent is labeled with α, and returns the label of its second parent if this is true or the label of its third parent if not. See [Poi95, BCSS98]) for formal details.

We say that a family C_n, $n \in \mathbb{N}$ of circuits is P-uniform if and only if there exists a polynomial time deterministic function describing each gate of each circuit.

The reader can find in [BCSS98] the definition of parallel machine over a structure $\mathcal{K}$. We will not give formal definitions here, since we will actually use the alternative characterization given by Proposition 2 below, proved in [BCSS98].

Proposition 2. *The $PAR_{\mathcal{K}}$ class of problems decidable in polynomial time by a parallel machine using an exponentially bounded number of processors is exactly the class of problem decidable by a P-uniform family of circuits of polynomial depth.*

The rest of this section is devoted to prove that, over any structure $\mathcal{K}$, class $\mathrm{PAR}_{\mathcal{K}}$ also corresponds to the class of decision functions definable with safe recursion with substitution.

5.2 Safe Recursion with Substitutions

Definition 3. *The set of functions defined with safe recursion with substitutions over $\mathbb{K}$ is the smallest set of functions: $(\mathbb{K}^*)^n \to \mathbb{K}^*$, containing the basic safe functions, and closed under the operations of safe composition and safe recursion with substitutions.*

Basic safe functions are defined in Section 3, as well as the operation of safe composition. We only need to define the notion of safe recursion with substitution : Let $h_1, \ldots, h_k : \mathbb{K} \times (\mathbb{K}^*)^2 \to \mathbb{K}^*$ and $g_1, \ldots, g_k : (\mathbb{K})^2 \times (\mathbb{K}^*)^{kl+1} \to \mathbb{K}^*$. Let the following safe recursive functions $\sigma_{\imath,\jmath} : \mathbb{K}^* \to \mathbb{K}^*, 0 < \imath \leq k, 0 < \jmath \leq l$ for an arbitrary l, called substitution functions. These functions need to be instanciated in the scheme. Here we assume that the arguments of these substitution functions are all safe. Functions $f_1, \ldots, f_k : (\mathbb{K})^2 \times (\mathbb{K}^*)^2 \to \mathbb{K}^*$ can then be defined by safe recursion with substitutions:

$$f_1(\epsilon, \overline{z}; \overline{u}, \overline{y}), \ldots, f_k(\epsilon, \overline{z}; \overline{u}, \overline{y}) = h_1(\overline{z}; \overline{u}, \overline{y}), \ldots, h_k(\overline{z}; \overline{u}, \overline{y})$$

$$f_1(a.\overline{x}, \overline{z}; \overline{u}, \overline{y}) = \begin{cases} g_1\,(\overline{x}, \overline{z}; f_1(\overline{x}, \overline{z}; \sigma_{1,1}(;\overline{u}), \overline{y}), \ldots, f_1(\overline{x}, \overline{z}; \sigma_{1,l}(;\overline{u}), \overline{y}), \ldots \\ \qquad f_k(\overline{x}, \overline{z}; \sigma_{k,1}(;\overline{u}), \overline{y}), \ldots, f_k(\overline{x}, \overline{z}; \sigma_{k,l}(;\overline{u}), \overline{y}), \overline{y}) \\ \qquad\qquad \text{if } \forall \imath, \jmath\ f_\imath(\overline{x}, \overline{z}; \sigma_{\imath,\jmath}(;\overline{u}), \overline{y}) \neq \bot \\ \bot \qquad\qquad \text{otherwise} \end{cases}$$

$$\vdots$$

$$f_k(a.\overline{x}, \overline{z}; \overline{u}, \overline{y}) = \begin{cases} g_k\,(\overline{x}, \overline{z}; f_1(\overline{x}, \overline{z}; \sigma_{1,1}(;\overline{u}), \overline{y}), \ldots, f_1(\overline{x}, \overline{z}; \sigma_{1,l}(;\overline{u}), \overline{y}), \ldots \\ \qquad f_k(\overline{x}, \overline{z}; \sigma_{k,1}(;\overline{u}), \overline{y}), \ldots, f_k(\overline{x}, \overline{z}; \sigma_{k,l}(;\overline{u}), \overline{y}), \overline{y}) \\ \qquad\qquad \text{if } \forall \imath, \jmath\ f_\imath(\overline{x}, \overline{z}; \sigma_{\imath,\jmath}(;\overline{u}), \overline{y}) \neq \bot \\ \bot \qquad\qquad \text{otherwise} \end{cases}$$

5.3 Simulation of a P-uniform Family of Circuits

By hypothesis, the family of circuits we want to simulate here is P-uniform. This means that there exists a polynomial time deterministic function which, given n the length of the input of the circuit and m the gate number, gives the description of the mth gate of circuit C_n.

We detail now how a gate is described:

- The single (remember, we simulate a decision function) output node is numbered 0. It is represented by $\epsilon \in \mathbb{K}^*$.
- Let r be the maximal arity of a relation or a function of the structure. Let $s = \lceil \lg(\max\{r, 3\}) \rceil$ be the size necessary to write the binary encoding of $0, 1, \ldots, \max\{r, 3\}$, the maximum number of parents for a given node. Assume that a gate is represented by $\overline{y}$. Its parents nodes, from its first to

its m^{th}, are represented by $\overline{a_0}.\overline{y}, \ldots, \overline{a_{m-1}}.\overline{y}$, where $\overline{a_\imath} \in \{\beta, \alpha\}^*$ represents the binary encoding of $\imath$ in $\mathbb{K}^*$, β being put in place of 0 and α in place of 1. We add as many βs as needed in front such that these $a_\imath$ have length s.

We define the safe recursive functions $\sigma_\imath$ such that $\sigma_\imath(\overline{y}) = a_\imath.\overline{y}$ where $a_\imath$ is defined above.

Note 4. We assume here a "tree" viewpoint for the description of the circuit, by opposition to the (more classical) "Directed Acyclic Graph" (DAG) viewpoint. In this tree viewpoint, the representations of a gate are codifications of the paths from the output node to it. In particular, a gate may have several representations. P-uniformity ensures that the translation can be done in polynomial time by a deterministic function.

Theorem 2 proved before in this paper ensures that this description can be computed with safe recursive functions. Therefore, we assume that we have the following safe recursive function $Gate$, which returns a code for the label of the node $\overline{y}$ in the circuit C_n, where $n = |\overline{x}|$ is the size of the input, and $\overline{x} = x_1 \ldots . x_n$:

$$Gate(\overline{x}; \overline{y}) = \begin{cases} \beta.x_\imath & \text{for an input gate corresponding} \\ & \text{to the } \imath^{th} \text{ coordinate of the input} \\ \alpha^\imath & \text{for a gate labeled with op}_\imath \\ \alpha^{k+\imath} & \text{for a gate labeled with rel}_\imath \\ \alpha^{k+l+1} & \text{for a } \textit{test} \text{ node} \end{cases}$$

Remember the functions $Equal_k$ defined in Section 4, and denote with $\overline{t}$, the current depth in the simulation of the circuit. The simulation of the circuit is done with the function $Eval$ defined as follows, where $k_\imath$ is the arity of $op_\imath$ and $l_\imath$ the arity of $rel_\imath$:

$$\begin{aligned} Eval(\epsilon, \overline{x}; \overline{y}) &= \mathrm{C}(; Gate(\overline{x}; \overline{y}), \mathrm{tl}(; Gate(\overline{x}; \overline{y})), \epsilon) \\ Eval(t_1.\overline{t}, \overline{x}; \overline{y}) &= \mathrm{C}\left(; Equal_1(; Gate(\overline{x}; \overline{y})), \mathrm{Op}_1\ (; Eval(\overline{t}, \overline{x}; \sigma_0(; \overline{y})), \right. \\ &\qquad \left. \ldots, Eval(\overline{t}, \overline{x}; \sigma_{k_1}(; \overline{y}))\right), \\ &\quad \vdots \\ &\quad \ldots \mathrm{C}\left(; Equal_{k+1}(; Gate(\overline{x}; \overline{y})), \mathrm{Rel}_1\ (; Eval(\overline{t}, \overline{x}; \sigma_0(; \overline{y})), \right. \\ &\qquad \left. \ldots, Eval(\overline{t}, \overline{x}; \sigma_{l_1}(; \overline{y}))\right), \\ &\quad \vdots \\ &\quad \ldots \mathrm{C}\left(; Equal_{k+l}(; Gate(\overline{x}; \overline{y})), \mathrm{Rel}_l\ (; Eval(\overline{t}, \overline{x}; \sigma_0(; \overline{y})), \right. \\ &\qquad \left. \ldots, Eval(\overline{t}, \overline{x}; \sigma_{l_l}(; \overline{y}))\right), \\ &\quad \mathrm{C}(; Equal_{k+l+1}(; Gate(\overline{x}; \overline{y})), \\ &\quad \mathrm{C}\left(Eval(\overline{t}, \overline{x}; \sigma_0(\overline{y};)), Eval(\overline{t}, \overline{x}; \sigma_1(\overline{y};)), Eval(\overline{t}, \overline{x}; \sigma_2(\overline{y};))\right), \\ &\qquad \mathrm{C}(; Gate(\overline{x}; \overline{y}), \mathrm{tl}(; Gate(\overline{x}; \overline{y})), \epsilon))) \ldots) \ldots)) \end{aligned}$$

Assume $p(n) = cn^d$ is a polynomial bounding the depth of the circuit C_n. The evaluation of C_n on input $\overline{x}$ of length n is then given by $Eval(\overline{t}, \overline{x}; \epsilon)$ where $|\overline{t}| = cn^d$. Lemma 2 gives the existence of a safe recursive function $P_{c,d}$ such that $|P_{c,d}(\overline{x};)| = cn^d$. The evaluation of the P-uniform family of circuits $\mathrm{C}_n, n \in \mathbb{N}$ is then given by the function $Circuit(\overline{x};) = Eval(P_{c,d}(\overline{x};), \overline{x}; \epsilon)$ defined with safe recursion with substitutions.

5.4 Evaluation of a Function Defined with Safe Recursion with Substitutions

Let f be a function defined with safe recursion with substitutions, and denote by f_n the restriction of f on the set of inputs of size at most n. We need to prove that f can be simulated by a P-uniform family of circuits $\mathcal{C}(f)_n, n \in \mathbb{N}$ of polynomial depth, each $\mathcal{C}(f)_n$ simulating f_n.

Let us prove by induction on the definition tree of f the following lemma:

Lemma 3. *For any function $f : (\mathbb{K}^*)^r \times (\mathbb{K}^*)^s$ defined with safe recursion with substitutions, let us denote by $D^{\ulcorner} f(\ldots)^{\urcorner}$ the depth of the circuit $\mathcal{C}_n$ simulating $f(\ldots)$ on inputs of size at most n. Then,*

$$\begin{aligned} &D^{\ulcorner} f(\overline{x_1}, \ldots, \overline{x_r}; \overline{y_1}, \ldots, \overline{y_s})^{\urcorner} \\ &\leq p_f(\max\{D^{\ulcorner} \overline{x_1}^{\urcorner}, \ldots, D^{\ulcorner} \overline{x_r}^{\urcorner}\}) + \max\{D^{\ulcorner} \overline{y_1}^{\urcorner}, \ldots, D^{\ulcorner} \overline{y_s}^{\urcorner}\} \end{aligned}$$

for some polynomial p_f, and

$$|f(\overline{x_1}, \ldots, \overline{x_r}; \overline{y_1}, \ldots, \overline{y_s})| \leq q_f(|\overline{x_1}| + \ldots + |\overline{x_s}|)$$

for some polynomial q_f.

Proof.

- If f is a basic function, the result is straightforward.
- If f is defined with the operation of safe composition:

$$\begin{aligned} &f(\overline{x_1} \ldots, \overline{x_r}, \overline{y_1}, \ldots, \overline{y_{p_1}}, \overline{z_1}, \ldots, \overline{z_{p_2}}; \overline{t_1}, \ldots, \overline{t_m}) \\ &= g(h_1(\overline{x_1}, \ldots, \overline{x_r}, \overline{y_1}, \ldots, \overline{y_{p_1}};); h_2(\overline{x_1}, \ldots, \overline{x_n}, \overline{z_1}, \ldots, \overline{z_{p_2}}; \overline{t_1}, \ldots, \overline{t_m})) \end{aligned}$$

  then, $\mathcal{C}(f)_n$ is the obtained by plugging $\mathcal{C}(h_1)_n$ and $\mathcal{C}(h_2)_n$ in the input nodes of $\mathcal{C}(g)_{q_{h_1}(n)+q_{h_2}(n)}$. Thus, when we apply the induction hypothesis to g:

$$\begin{aligned} p_f(n) &\leq p_g\left(p_{h_1}(n)\right) + \max\{p_{h_1}(n), p_{h_2}(n)\} \\ q_f(n) &\leq q_g\left(q_{h_1}(n)\right) \end{aligned}$$

- If f is defined with the operation of safe recursion with substitutions:
- The non-trivial case is the case of a function f defined with safe recursion, as in Definition 3.

Let us apply induction hypothesis to function $g_\imath$ in the expression $f_\imath(a.\overline{x},\overline{z};\overline{u},\overline{y})$. $\overline{u}$ and $\overline{y}$ are given by their respective circuits plugged at the right position. The depth of the circuit evaluating

$$f_\imath(a.\overline{x},\overline{z};\overline{u},\overline{y}) = g_\imath\left(\overline{x},\overline{z}; f_1(\overline{x},\overline{z};\sigma_{1,1}(;\overline{u}),\overline{y}),\ldots,f_1(\overline{x},\overline{z};\sigma_{1,l}(;\overline{u}),\overline{y}),\ldots\right.$$
$$\left. f_k(\overline{x},\overline{z};\sigma_{k,1}(;\overline{u}),\overline{y}),\ldots,f_k(\overline{x},\overline{z};\sigma_{k,l}(;\overline{u}),\overline{y}),\overline{y}\right)$$

is given by $p_{g_\imath}(\max\{D^\ulcorner\overline{x}^\urcorner, D^\ulcorner\overline{z}^\urcorner\}) + \max_{\imath,\jmath}\{D^\ulcorner f_\imath(\overline{x},\overline{z};\sigma_{\imath,\jmath}(;\overline{u}),\overline{y})^\urcorner, D^\ulcorner\overline{y}^\urcorner\}$. Assume: for any $\imath$, $p_{g_\imath}$ is bounded by some polynomial p_g. Assume moreover: for any $\imath,\jmath$, $p_{\sigma_{\imath,\jmath}}$ is bounded by some p_σ. We get:

$$\begin{aligned}
&D^\ulcorner f(a.\overline{x},\overline{z};\overline{u},\overline{y})^\urcorner\\
&\leq D^\ulcorner\overline{y}^\urcorner + p_g(\max\{D^\ulcorner\overline{x}^\urcorner, D^\ulcorner\overline{z}^\urcorner\}) + \max_{\imath,\jmath}\{D^\ulcorner f_\imath(\overline{x},\overline{z};\sigma_{\imath,\jmath}(;\overline{u}),\overline{y})^\urcorner\}\\
&\leq D^\ulcorner\overline{y}^\urcorner + p_g(\max\{D^\ulcorner\overline{x}^\urcorner, D^\ulcorner\overline{z}^\urcorner\}) + p_g(\max\{D^\ulcorner\overline{\mathrm{tl}(x)}^\urcorner, D^\ulcorner\overline{z}^\urcorner\}) + \ldots\\
&\quad \ldots + p_g(\max D^\ulcorner\overline{\epsilon}^\urcorner, D^\ulcorner\overline{z}^\urcorner\}) + p_h(D^\ulcorner\overline{z}^\urcorner) + \max\{|a.\overline{x}|p_\sigma() + D^\ulcorner\overline{u}^\urcorner, D^\ulcorner\overline{y}^\urcorner\}
\end{aligned}$$

Assuming without loss of generality p_g monotone, we get

$$\begin{aligned}
D^\ulcorner f(a.\overline{x},\overline{z};\overline{u},\overline{y})^\urcorner \leq |a.\overline{z}|p_g(\max\{D^\ulcorner\overline{x}^\urcorner, D^\ulcorner\overline{z}^\urcorner\})\\
D^\ulcorner\overline{y}^\urcorner + p_h(D^\ulcorner\overline{z}^\urcorner) + \max\{|a.\overline{x}|p_\sigma() + D^\ulcorner\overline{u}^\urcorner, D^\ulcorner\overline{y}^\urcorner\},
\end{aligned}$$

from which the result follows for p_f. For q_f, we need:
$|f_\imath(a.\overline{x},\overline{z},\overline{u};\overline{y})| \leq q_{g_\imath}\left(|\overline{x}| + |\overline{x}|\right)$. This ends the proof of our lemma.

It follows from the lemma that every circuit of the family $\mathcal{C}(f)_n$ has a polynomial depth in n. The P-uniformity is given by the description of the circuit as above.

In the classical setting (see [LM95]), safe recursion with substitution characterizes the class PSPACE. However, in the general setting, this notion of working space is meaningless, as pointed in [Mic89]: on some structures like $(\mathbb{R},0,1,=,+,-,*)$, any computation can be done in constant working space. However, since we have in the classical setting PAR = PSPACE, our result extends the classical one from [LM95].

References

[BC92] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[BCSS98] Lenore Blum, Felipe Cucker, Michael Shub, and Steve Smale. *Complexity and Real Computation*. Springer Verlag, 1998.

[BMdN02] Olivier Bournez, Jean-Yves Marion, and Paulin de Naurois. Safe recursion over an arbitrary structure: Deterministic polynomial time. Technical report, LORIA, 2002.

[BSS89] Lenore Blum, Mike Shub, and Steve Smale. On a theory of computation and complexity over the real numbers: Np-completeness, recursive functions, and universal machines. *Bulletin of the American Mathematical Society*, 21:1–46, 1989.

[Clo95] P. Clote. Computational models and function algebras. In D. Leivant, editor, *LCC'94*, volume 960, pages 98–130, 1995.

[Cob62] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam, 1962.

[Coo92] S. A. Cook. Computability and complexity of higher-type functions. In Y. Moschovakis, editor, *Logic from Computer Science*, pages 51–72. Springer-Verlag, New York, 1992.

[CSS94] F. Cucker, M. Shub, and S. Smale. Separation of complexity classes in Koiran's weak model. *Theoretical Computer Science*, 133(1):3–14, 11 October 1994.

[Cuc92] F. Cucker. $\mathrm{P}_{\mathbb{R}} \neq \mathrm{NC}_{\mathbb{R}}$. *Journal of Complexity*, 8:230–238, 1992.

[EF95] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite Model Theory*. Perspectives in Mathematical Logic. Springer-Verlag, Berlin, 1995.

[Fag74] R. Fagin. Generalized first order spectra and polynomial time recognizable sets. In R. Karp, editor, *Complexity of Computation*, pages 43–73. SIAM-AMS, 1974.

[Goo94] J. B. Goode. Accessible telephone directories. *The Journal of Symbolic Logic*, 59(1):92–105, March 1994.

[Gur83] Y. Gurevich. Algebras of feasible functions. In *Twenty Fourth Symposium on Foundations of Computer Science*, pages 210–214. IEEE Computer Society Press, 1983.

[Hof99] M. Hofmann. Type systems for polynomial-time computation, 1999. Habilitation.

[Imm99] N. Immerman. *Descriptive Complexity*. Springer, 1999.

[Jon00] N. Jones. The expressive power of higher order types or, life without cons. 2000.

[Lei95] D. Leivant. Intrinsic theories and computational complexity. In *LCC'94*, number 960, pages 177–194, 1995.

[LM93] D. Leivant and J-Y Marion. Lambda calculus characterizations of poly-time. *fi*, 19(1,2):167,184, September 1993.

[LM95] Daniel Leivant and Jean-Yves Marion. Ramified recurrence and computational complexity II: substitution and poly-space. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic, 8th Workshop, CSL'94*, volume 933 of *Lecture Notes in Computer Science*, pages 369–380, Kazimierz, Poland, 1995. Springer.

[Mee92] K. Meer. A note on a $P \neq NP$ result for a restricted class of real machines. *Journal of Complexity*, 8:451–453, 1992.

[Mic89] Christian Michaux. Une remarque à propos des machines sur $\mathbb{R}$ introduites par Blum, Shub et Smale. In *C. R. Acad. Sc. de Paris*, volume 309 of *1*, pages 435–437. 1989.

[MM00] J-Y Marion and J-Y Moyen. Efficient first order functional program interpreter with time bound certifications. In *LPAR*, volume 1955, pages 25–42. Springer, Nov 2000.

[Poi95] Bruno Poizat. *Les petits cailloux*. aléas, 1995.

[RIR01] B. Kapron R. Irwin and J. Royer. On characterizations of the basic feasible functionals. 11:117–153, 20001.

[Saz80] V. Sazonov. Polynomial computability and recursivity in finite domains. *Elektronische Informationsverarbeitung und Kybernetik*, 7:319–323, 1980.

An Intrinsic Characterization of Approximate Probabilistic Bisimilarity

Franck van Breugel[1], Michael Mislove[2], Joël Ouaknine[3], and James Worrell[2]

[1] York University, Department of Computer Science
4700 Keele Street, Toronto, M3J 1P3, Canada
[2] Tulane University, Department of Mathematics
6823 St Charles Avenue, New Orleans LA 70118, USA
[3] Computer Science Department, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh PA 15213, USA

Abstract. In previous work we have investigated a notion of approximate bisimilarity for labelled Markov processes. We argued that such a notion is more realistic and more feasible to compute than (exact) bisimilarity. The main technical tool used in the underlying theory was the Hutchinson metric on probability measures. This paper gives a more fundamental characterization of approximate bisimilarity in terms of the notion of (exact) similarity. In particular, we show that the topology of approximate bisimilarity is the Lawson topology with respect to the simulation preorder. To complement this abstract characterization we give a statistical account of similarity, and by extension, of approximate bisimilarity, in terms of the process testing formalism of Larsen and Skou.

1 Introduction

Labelled Markov processes provide a simple operational model of reactive probabilistic systems. A *labelled Markov process* consists of a measurable space (X, Σ) of states, a family Act of actions, and a transition probability function $\mu_{-,-}$ that, given a state $x \in X$ and an action $a \in \text{Act}$, yields the probability $\mu_{x,a}(A)$ that the next state of the process will be in the measurable set $A \in \Sigma$ after performing action a in state x. These systems are a generalization of the probabilistic labelled transition systems with discrete distributions considered by Larsen and Skou [16].

The basic notion of process equivalence in concurrency is bisimilarity. This notion, due to Park [18], asserts that processes are *bisimilar* iff any action by

[1] Supported by the Natural Sciences and Engineering Research Council of Canada.
[2] Supported by the US National Science Foundation and the US Office of Naval Research (ONR).
[3] Supported by ONR contract N00014-95-1-0520, Defense Advanced Research Project Agency and the Army Research Office under contract DAAD19-01-1-0485.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 200–215, 2003.

either can be matched with the same action by the other, and the resulting processes are also bisimilar. Larsen and Skou adapted the notion of bisimilarity to discrete probabilistic systems, by defining an equivalence relation R on states to be a bisimulation if related states have *exactly matching* probabilities of making transitions into any R-equivalence class. Later the theory of probabilistic bisimilarity was extended beyond the discrete setting by Edalat, Desharnais and Panangaden [8]. From quite early on, however, it was realized that for probabilistic systems a notion of approximate bisimilarity might prove more appropriate than a notion of exact bisimilarity. One advantage of such a notion is that it is more informative: one can say that two processes are almost bisimilar, even though they do not behave exactly the same. More fundamentally, one could even argue that the idea of exact bisimilarity is meaningless if the probabilities appearing in the model of a system are approximations based on statistical data, or if the algorithm used to calculate bisimilarity is not based on exact arithmetic.

Desharnais, Gupta, Jagadeesan and Panangaden [9] formalized a notion of approximate bisimilarity by defining a metric[1] on the class of labelled Markov processes. Intuitively the smaller the distance between two processes, the more alike their behaviour; in particular, they showed that states are at zero distance just in case they are bisimilar. The original definition of the metric in [9] was stated through a real-valued semantics for a variation of Larsen and Skou's probabilistic modal logic [16]. Later it was shown how to give a coinductive definition of this metric using the Hutchinson metric on probability measures [4]. Using this characterization [5] gave an algorithm based on linear programming to approximate the distance between the states of a finite labelled Markov process.

The fact that zero distance coincides with bisimilarity can be regarded as a sanity check on the definition of the metric. The papers [9,4] also feature a number of examples showing how processes with similar transition probabilities are close to one another. A more precise account of how the metric captures approximate bisimilarity is given in [6], where it is shown that convergence in the metric can be characterized in terms of the convergence of observable behaviour; the latter is formalized by Larsen and Skou's process testing formalism [16]. As Di Pierro, Hankin and Wiklicky [20] argue, such an account is vital if one wants to use the metric to generalize the formulations of probabilistic non-interference based on bisimilarity.

Both of the above mentioned characterizations of the metric for approximate bisimilarity are based on the idea of defining a distance between measures by integration against a certain class of functions, which is a standard approach from functional analysis. But it is reasonable to seek an intrinsic characterization of approximate bisimilarity that does not rely on auxiliary notions such as integration. In this paper we give such a characterization. We show that the topology induced by the metric described above coincides with the Lawson topology on the domain that arises by endowing the class of labelled Markov processes with the probabilistic *simulation* preorder. The characterization is intrinisic: the Lawson

[1] Strictly speaking, a pseudometric since distinct processes can have distance zero.

topology is defined solely in terms of the order on the domain. For this reason, we view this characterization as more fundamental than the existing ones.

Our results are based on a simple interaction between domain theory and measure theory. This is captured in Corollary 2 which shows that the Lawson topology on the probabilistic powerdomain of a coherent domain agrees with the weak topology on the family of subprobability measures on the underlying coherent domain, itself endowed with the Lawson topology. A simple corollary of this result is that the probabilistic powerdomain of a coherent domain is again coherent, a result first proved by Jung and Tix [15] using purely domain-theoretic techniques.

We use the coincidence of the Lawson and weak topologies to analyze a recursively defined domain D of probabilistic processes first studied by Desharnais *et al.* [10]. The key property of the domain D is that it is equivalent (as a preordered class) to the class of all labelled Markov processes equipped with the simulation preorder. The proof of this result in [10] makes use of a discretization construction, which shows how an arbitrary labelled Markov process can be recovered as the limit of a chain of finite state approximations. In this paper, we give a more abstract proof: we use the coincidence of the Lawson and weak topologies to show that the domain D has a universal property: namely, it is final in a category of labelled Markov processes.

A minor theme of the present paper is to extend the characterization of approximate bisimilarity in terms of the testing formalism of Larsen and Skou [16]. We show that bisimilarity can be characterized as testing equivalence, where one records only positive observations of tests. On the other hand, characterizing similarity requires one also to record negative observations, i.e., refusals of actions.

2 Labelled Markov Processes

Let us assume a fixed set Act of actions. For ease of exposition we suppose that Act is finite, but all our results hold in case it is countable.

Definition 1. *A* labelled Markov process *is a triple* $\langle X, \Sigma, \mu \rangle$ *consisting of a set* X *of states, a* σ*-field* Σ *on* X*, and a transition probability function* $\mu : X \times \mathrm{Act} \times \Sigma \to [0,1]$ *such that*

1. *for all* $x \in X$ *and* $a \in \mathrm{Act}$*, the function* $\mu_{x,a}(\cdot) : \Sigma \to [0,1]$ *is a subprobability measure, and*
2. *for all* $a \in \mathrm{Act}$ *and* $A \in \Sigma$*, the function* $\mu_{-,a}(A) : X \to [0,1]$ *is measurable.*

The function $\mu_{-,a}$ describes the reaction of the process to the action a selected by the environment. This represents a reactive model of probabilistic processes. Given that the process is in state x and reacts to the action a chosen by the environment, $\mu_{x,a}(A)$ is the probability that the process makes a transition to a state in the set of states A. Note that we consider *sub*probability measures, i.e. positive measures with total mass no greater than 1, to allow for the possibility

that the process may refuse an action. The probability of refusal of the action a given the process is in state x is $1 - \mu_{x,a}(X)$.

An important special case is when the σ-field Σ is the powerset of X and, for all actions a and states x, the subprobability measure $\mu_{x,a}(\cdot)$ is completely determined by a discrete subprobability distribution. This case corresponds to the original probabilistic transition system model of Larsen and Skou [16].

A natural notion of a map between labelled Markov processes is given in:

Definition 2. *Given labelled Markov processes $\langle X, \Sigma, \mu\rangle$ and $\langle X', \Sigma', \mu'\rangle$, a measurable function $f\colon X \to X'$ is a* zigzag map *if for all $x \in X$, $a \in$ Act and $A' \in \Sigma'$, $\mu_{x,a}(f^{-1}(A')) = \mu'_{f(x),a}(A)$.*

Probabilistic bisimulations (henceforth just bisimulations) were first introduced in the discrete case by Larsen and Skou [16]. They are the relational counterpart of zigzag maps and can also be seen, in a very precise way, as the probabilistic analogues of the strong bisimulations of Park and Milner [17]. The definition of bisimulation was extended to labelled Markov processes in [8,10].

Definition 3. *Let $\langle X, \Sigma, \mu\rangle$ be a labelled Markov process. A reflexive relation R on X is a* simulation *if whenever $x\,R\,y$, then for all $a \in$ Act and all R-closed $A \in \Sigma$, $\mu_{x,a}(A) \leqslant \mu_{y,a}(A)$. A set A is R-*closed *if $x \in A$ and $x\,R\,y$ imply $y \in A$. We say that R is a* bisimulation *if, in addition, whenever $x\,R\,y$ then $\mu_{x,a}(X) = \mu_{y,a}(X)$. Two states are* bisimilar *if they are related by some bisimulation.*

The notions of simulation and bisimulation are very close in the probabilistic case. The extra condition $\mu_{x,a}(X) = \mu_{y,a}(X)$ in the definition of bisimulation allows one to show that if R is a bisimulation, then the inverse relation R^{-1} is also a bisimulation. It follows that the union of all bisimulations is an equivalence relation R such that xRy implies $\mu_{x,a}(A) = \mu_{y,a}(A)$ for all $a \in$ Act and measurable R-closed $A \subseteq X$. This equality, which entails infinite precision, is the source of the fragility in the definition of bisimilarity. This motivates the idea of defining a notion of approximate bisimilarity.

2.1 A Metric for Approximate Bisimilarity

We recall a variant of Larsen and Skou's probabilistic modal logic [16], and a real-valued semantics due to Desharnais *et al.* [9]. The set of formulas of probabilistic modal logic (PML), denoted $\mathcal{F}$, is given by the following grammar:

$$f ::= \top \mid f \wedge f \mid f \vee f \mid \langle a\rangle f \mid f \dot{-} q$$

where $a \in$ Act and $q \in [0,1] \cap \mathbb{Q}$.

The modal connective $\langle a\rangle$ and truncated subtraction replace a single connective $\langle a\rangle_q$ in Larsen and Skou's presentation.

Fix a constant $0 < c < 1$ once and for all. Given a labelled Markov process $\langle X, \Sigma, \mu\rangle$, a formula f determines a measurable function $f\colon X \to [0,1]$ according

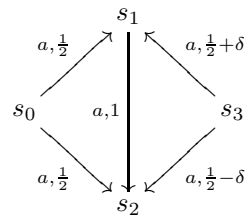
to the following rules. $\top$ is interpreted as the constant function 1, conjunction and disjunction are interpreted as max and min respectively, truncated subtraction is defined in the obvious manner, and $(\langle a \rangle f)(x) = c \int f d\mu_{x,a}$ for each $a \in \mathrm{Act}$. Thus the interpretation of a formula f depends on c. The role of this constant is to discount observations made at greater modal depth.

Given a labelled Markov process $\langle X, \Sigma, \mu \rangle$, one defines a metric d on X by

$$d(x, y) = \sup_{f \in \mathcal{F}} |f(x) - f(y)| .$$

It is shown in [9] that zero distance in this metric coincides with bisimilarity. Roughly speaking, the smaller the distance between states, the closer their behaviour. The exact distance between two states depends on the value of c, but one consequence of our results is that the topology induced by the metric d is independent of the original choice of c.

Example 1. In the labelled Markov process below, $d(s_0, s_3) = c^2\delta$. The two states are bisimilar just in case $\delta = 0$.



3 Domain Theory

A *directed subset* $A \subseteq D$ of a poset D is one for which every finite subset of A has an upper bound in A, and a *directed complete partial order (dcpo)* is a poset D in which each directed set A has a least upper bound, denoted $\sqcup A$. If D is a dcpo, and $x, y \in D$, then we write $x \ll y$ if each directed subset $A \subseteq D$ with $y \sqsubseteq \sqcup A$ satisfies $\uparrow x \cap A \neq \emptyset$. We then say x *is way-below* y. Let $\mathop{\downarrow} y = \{x \in D \mid x \ll y\}$; we say that D is *continuous* if it has a *basis*, i.e., a subset $B \subseteq D$ such that for each $y \in D$, $\mathop{\downarrow} y \cap B$ is directed with supremum y. We use the term *domain* to mean a continuous dcpo.

A subset U of a dcpo D is *Scott-open* if it is an upper set (i.e., $U = \uparrow U$) and for each directed set $A \subseteq D$, if $\sqcup A \in U$ then $A \cap U \neq \emptyset$. The collection ΣD of all Scott-open subsets of D is called the *Scott topology* on D. If D is continuous, then the Scott topology on D is locally compact, and the sets $\Uparrow x$ where $x \in D$ form a basis for this topology. Given dcpos D and E, a function $f \colon D \to E$ is continuous with respect to the Scott topologies on D and E iff it is monotone and preserves directed suprema: for each directed $A \subseteq D$, $f(\sqcup A) = \sqcup f(A)$.

Another topology of interest on a continuous dcpo D is the *Lawson topology*. This topology is the join of the Scott topology and the *lower interval topology*, where the latter is generated by sub-basic open sets of the form $D \setminus \uparrow x$. Thus, the

Lawson topology has the family $\{\uparrow\!\!\!\uparrow x \setminus \uparrow F \mid x \in D, F \subseteq D \text{ finite}\}$ as a basis. The Lawson topology on a domain is always Hausdorff. A domain which is compact in its Lawson topology is called *coherent*.

4 The Probabilistic Powerdomain

We briefly recall some basic definitions and results about valuations and the probabilistic powerdomain.

Definition 4. *Let (X, Ω) be a topological space. A* valuation *on X is a mapping $\mu\colon \Omega \to [0,1]$ satisfying:*

1. $\mu\emptyset = 0$.
2. $U \subseteq V \Rightarrow \mu U \leqslant \mu V$.
3. $\mu(U \cup V) + \mu(U \cap V) = \mu U + \mu V, \;\; U, V \in \Omega$

Departing from standard practice, we also require that a valuation is Scott continuous as a map $(\Omega, \subseteq) \to ([0,1], \leqslant)$.

Each element $x \in X$ gives rise to a valuation δ_x defined by $\delta_x(U) = 1$ if $x \in U$, and $\delta_x(U) = 0$ otherwise. A *simple valuation* has the form $\sum_{a\in A} r_a \delta_a$ where A is a finite subset of X, $r_a \geqslant 0$, and $\sum_{a\in A} r_a \leqslant 1$.

We write $\mathbb{V}X$ for the space whose points are valuations on X, and whose topology is generated by sub-basic open sets of the form $\{\mu \mid \mu U > r\}$, where $U \in \Omega$ and $r \in [0,1]$. The specialization order[2] on $\mathbb{V}X$ with respect to this topology is given by $\mu \sqsubseteq \mu'$ iff $\mu U \leqslant \mu' U$ for all $U \in \Omega$. $\mathbb{V}$ extends to an endofunctor on Top – the category of topological spaces and continuous maps – by defining $\mathbb{V}(f)(\mu) = \mu \circ f^{-1}$ for a continuous map f.

Suppose D is a domain regarded as a topological space in its Scott topology. Jones [14] has shown that the specialization order defines a domain structure on $\mathbb{V}D$, with the set of simple valuations forming a basis. Furthermore, it follows from the following proposition that the topology on $\mathbb{V}D$ is actually the Scott topology with respect to the pointwise order on valuations.

Proposition 1 (Edalat [11]). *A net $\langle \mu_\alpha \rangle$ converges to μ in the Scott topology on $\mathbb{V}D$ iff $\liminf \mu_\alpha U \geqslant \mu U$ for all Scott open $U \subseteq D$.*

Finally, Jung and Tix [15] have shown that if D is a coherent domain then so is $\mathbb{V}D$. In summary we have the following proposition.

Proposition 2. *The endofunctor $\mathbb{V}\colon \mathsf{Top} \to \mathsf{Top}$ preserves the subcategory $\omega\mathsf{Coh}$ of coherent domains with countable bases equipped with their Scott topologies.*

The fact that we define the functor $\mathbb{V}$ over Top rather than just considering the probabilistic powerdomain as a construction on domains has a payoff later on.

[2] The specialization preorder on a topological space is defined by $x \sqsubseteq y$ iff, for every open set U, $x \in U$ implies $y \in U$. It is a partial order precisely when the space is T_0.

Obviously, valuations bear a close resemblance to measures. In fact, any valuation on a coherent domain D may be uniquely extended to a measure on Borel σ-field generated by the Scott topology (equivalently by the Lawson topology) on D [2]. Thus we may consider the so-called *weak topology* on $\mathbb{V}D$. This is the weakest topology such that for each Lawson continuous function $f\colon D \to [0,1]$, $\Phi_f(\mu) = \int f d\mu$ defines a continuous function $\Phi_f\colon \mathbb{V}D \to [0,1]$. Alternatively, it may be characterized by saying that a net of valuations $\langle \mu_\alpha \rangle$ converges to μ iff $\liminf \mu_\alpha O \geqslant \mu O$ for each Lawson open set O (cf. [19, Thm II.6.1]). We emphasize that the weak topology on $\mathbb{V}D$ is defined with respect to the Lawson topology on D.

5 The Lawson Topology on $\mathbb{V}D$

In this section we show that for a coherent domain D, the Lawson topology on $\mathbb{V}D$ coincides with the weak topology.

Proposition 3. *[Jones [14]] Suppose $\mu \in \mathbb{V}D$ is an arbitrary valuation, then $\sum_{a\in A} r_a \delta_a \sqsubseteq \mu$ iff $(\forall B \subseteq A) \sum_{a\in B} r_a \leqslant \mu(\uparrow B)$.*

Proposition 4. *Let $F = \{x_1, ..., x_n\} \subseteq D$, $\mu \in \mathbb{V}D$ and $\varepsilon > 0$ be given. Then there exists a finite set $\mathcal{K}$ of simple valuations such that $\mu \notin \uparrow \mathcal{K}$ but if $\nu \in \mathbb{V}D$ satisfies $\nu(\uparrow F) > \mu(\uparrow F) + \varepsilon$ then $\nu \in \uparrow \mathcal{K}$.*

Proof. Let $\delta = \varepsilon/n$. Define $f_\delta\colon [0,1] \to [0,1]$ by $f_\delta(x) = \max\{m\delta \mid m\delta \ll x, m \in \mathbb{N}\}$. Next we define $\mathcal{K}$ to be the finite set

$$\mathcal{K} = \left\{ \sum_{i=1}^{n} r_i \delta_{x_i} \mid \mu(\uparrow F) < \sum_{i=1}^{n} r_i \leqslant 1 \text{ and } \{r_1, ..., r_n\} \subseteq \operatorname{Ran} f_\delta \right\}.$$

From Proposition 3 we immediately deduce $\mu \notin \uparrow \mathcal{K}$. Now given $\nu \in \mathbb{V}D$ with $\nu(\uparrow F) > \mu(\uparrow F) + \varepsilon$, we set $r_i = f_\delta(\nu(\uparrow x_i \setminus \bigcup_{j<i} \uparrow x_j))$ for $i \in \{1, ..., n\}$. First we verify that $\sum_{i=1}^{m} r_i \delta_{x_i} \in \mathcal{K}$. Now

$$\begin{aligned} \nu(\uparrow F) - \sum_{i=1}^{n} r_i &= \nu(\uparrow F) - \sum_{i=1}^{n} f_\delta(\nu(\uparrow x_i \setminus \bigcup_{j<i} \uparrow x_j)) \\ &= \sum_{i=1}^{n} \left(\nu(\uparrow x_i \setminus \bigcup_{j<i} \uparrow x_j) - f_\delta(\nu(\uparrow x_i \setminus \bigcup_{j<i} \uparrow x_j)) \right) \\ &< n\delta = \varepsilon. \end{aligned}$$

It follows that $\sum_{i=1}^{n} r_i > \mu(\uparrow F)$, thus $\sum_{i=1}^{n} r_i \delta_{x_i} \in \mathcal{K}$.

Finally, we observe that $\sum_{i=1}^{n} r_i \delta_{x_i} \sqsubseteq \nu$ since, if $B \subseteq \{1, ..., n\}$, then

$$\sum_{i\in B} r_i = \sum_{i\in B} f_\delta(\nu(\uparrow x_i \setminus \bigcup_{j<i} \uparrow x_j)) \leqslant \sum_{i\in B} \nu(\uparrow x_i \setminus \bigcup_{j<i} \uparrow x_j) \leqslant \nu(\uparrow B).$$

Proposition 5. *A net $\langle \mu_\alpha \rangle$ converges to μ in the lower interval topology on $\mathbb{V}D$ iff $\limsup \mu_\alpha E \leqslant \mu E$ for all finitely generated upper sets E.*

Proof. Suppose $\mu_\alpha \to \mu$. Let $E = \uparrow F$, where F is finite, and suppose $\varepsilon > 0$ is given. Then by Proposition 4 there is a finite set $\mathcal{K}$ of simple valuations such that $\mu \notin \uparrow \mathcal{K}$ and for all valuations ν, $\nu \notin \uparrow \mathcal{K}$ implies $\nu E \leqslant \mu E + \varepsilon$. Then we conclude that $\limsup \mu_\alpha E \leqslant \mu E + \varepsilon$ since the net μ_α is eventually in the open set $\mathbb{V}D \setminus \uparrow \mathcal{K}$.

Conversely, suppose $\mu_\alpha \not\to \mu$. Then μ has a sub-basic open neighbourhood $\mathbb{V}D \setminus \uparrow \rho$ such that some subnet μ_β never enters this neighbourhood. We can assume $\rho = \sum_{a \in A} r_a \delta_a$ is a simple valuation. Since $\rho \not\sqsubseteq \mu$ there exists $B \subseteq A$ such that $\sum_{a \in B} r_a > \mu(\uparrow B)$. But $\mu_\beta(\uparrow B) \geqslant \sum_{a \in B} r_a > \mu(\uparrow B)$ for all β. Thus $\limsup \mu_\alpha(\uparrow B) > \mu(\uparrow B)$. □

Corollary 1. *Let $\langle \mu_\alpha \rangle$ be a net in $\mathbb{V}D$. Then $\langle \mu_\alpha \rangle$ converges to μ in the Lawson topology on $\mathbb{V}D$ iff*

1. *$\liminf \mu_\alpha U \geqslant \mu U$ for all Scott open $U \subseteq D$.*
2. *$\limsup \mu_\alpha E \leqslant \mu E$ for all finitely generated upper sets $E \subseteq D$.*

Proof. Combine Propositions 1 and 5. □

Corollary 2. *If D is Lawson compact, then so is $\mathbb{V}D$ and the weak and Lawson topologies agree on $\mathbb{V}D$.*

Proof. Recall [19, Thm II.6.4] that the weak topology on the space of Borel measures on a compact space is itself compact. By Corollary 1, the Lawson topology on $\mathbb{V}D$ is coarser than the weak topology. But it is a standard fact that if a compact topology is finer than a Hausdorff topology then the two must coincide.

The Lawson compactness of $\mathbb{V}D$ was first proved by Jung and Tix in [15]. Their proof is purely domain theoretic and doesn't use the compactness of the weak topology.

6 A Final Labelled Markov Process

In a previous paper [4] we used the Hutchinson metric on probability measures to construct a final object in the category of labelled Markov processes and zigzag maps. Here we show that one may also construct a final labelled Markov process as a fixed point D of the probabilistic powerdomain. As we mentioned in the introduction, the significance of this result is that D can be used to represent the class of all labelled Markov processes in the simulation preorder.

Given a measurable space $X = \langle X, \Sigma \rangle$, we write $\mathbb{M}X$ for the set of subprobability measures on X. For each measurable subset $A \subseteq X$ we have a projection function $p_A \colon \mathbb{M}X \to [0,1]$ sending μ to μA. We take $\mathbb{M}X$ to be a measurable

space by giving it the smallest σ-field such that all the projections p_A are measurable. Next, $\mathbb{M}$ is turned into a functor $\mathsf{Mes} \to \mathsf{Mes}$, where Mes denotes the category of measure spaces and measurable maps, by defining $\mathbb{M}(f)(\mu) = \mu \circ f^{-1}$ for $f : X \to Y$ and $\mu \in \mathbb{M}X$; see Giry [12] for details.

Definition 5. *Let $\mathcal{C}$ be a category and $F : \mathcal{C} \to \mathcal{C}$ a functor. An* F-coalgebra *consists of an object C in $\mathcal{C}$ together with an arrow $f : C \to FC$ in $\mathcal{C}$. An* F-homomorphism *from F-coalgebra $\langle C, f\rangle$ to F-coalgebra $\langle D, g\rangle$ is an arrow $h : C \to D$ in $\mathcal{C}$ such that $Fh \circ f = g \circ h$. F-coalgebras and F-homomorphisms form a category whose final object, if it exists, is called the* final F-coalgebra.

Given a labelled Markov process $\langle X, \Sigma, \mu\rangle$, μ may be regarded as a measurable map $X \to \mathbb{M}(X)^{\mathrm{Act}}$. That is, labelled Markov processes are nothing but coalgebras of the endofunctor $\mathbb{M}(-)^{\mathrm{Act}}$ on the category Mes. Furthermore the coalgebra homomorphisms in this case are just the zigzag maps, cf. [8].

Next, we relate the functor $\mathbb{M}$ to the probabilistic powerdomain functor $\mathbb{V}$. To mediate between domains and measure spaces we introduce the forgetful functor $\mathbb{U} : \omega\mathsf{Coh} \to \mathsf{Mes}$ which maps a coherent domain to the Borel measurable space generated by the Scott topology (equivalently by the Lawson topology).

Proposition 6. *The forgetful functor $\mathbb{U} : \omega\mathsf{Coh} \to \mathsf{Mes}$ satisfies $\mathbb{M} \circ \mathbb{U} = \mathbb{U} \circ \mathbb{V}$.*

Proof. (Sketch) Given a coherent domain D with countable basis, since valuations on the Scott topology on D are in 1-1 correspondence with Borel measures on $\mathbb{U}(D)$, we have a bijection between the points of the measurable spaces $\mathbb{M}\mathbb{U}(D)$ and $\mathbb{U}\mathbb{V}(D)$. That this bijection is an isomorphism of measurable spaces follows from the coincidence of the Lawson and weak topologies and the unique structure theorem[3]. □

The following proposition is a straightforward adaptation of [19, Thm I.1.10].

Proposition 7. *The forgetful functor $\mathbb{U} : \omega\mathsf{Coh} \to \mathsf{Mes}$ preserves limits of ω^{op}-chains.*

Starting with the final object of $\omega\mathsf{Coh}$, we construct the chain

$$1 \xleftarrow{!} \mathbb{V}1 \xleftarrow{\mathbb{V}!} \mathbb{V}^2 1 \xleftarrow{\mathbb{V}^2!} \mathbb{V}^3 1 \xleftarrow{\mathbb{V}^3!} \cdots \tag{1}$$

by iterating the functor $\mathbb{V}$. Writing $\{\mathbb{V}^n 1 \xleftarrow{\pi_n} \mathbb{V}^\omega 1\}_{n<\omega}$ for the limit cone of this chain, there is a unique 'connecting' map $\mathbb{V}^\omega 1 \leftarrow \mathbb{V}\mathbb{V}^\omega 1$ whose composition with π_n gives $\mathbb{V}\pi_n$.

Proposition 8. (i) *The image of (1) under the forgetful functor $\mathbb{U} : \omega\mathsf{Coh} \to \mathsf{Mes}$ is equal to the chain*

$$1 \xleftarrow{!} \mathbb{M}1 \xleftarrow{\mathbb{M}!} \mathbb{M}^2 1 \xleftarrow{\mathbb{M}^2!} \mathbb{M}^3 1 \longleftarrow \cdots \tag{2}$$

similarly obtained by iterating the functor $\mathbb{M}$.

[3] In a Polish space any sub σ-field of the Borel σ-field which is countably generated and separates points is equal to the whole Borel σ-field [3].

(ii) *The forgetful functor* $\mathbb{U}\colon \omega\mathsf{Coh} \to \mathsf{Mes}$ *maps* $\mathbb{V}^\omega 1$ *to* $\mathbb{M}^\omega 1$.
(iii) *The image of the connecting map* $\mathbb{V}^\omega 1 \leftarrow \mathbb{V}(\mathbb{V}^\omega 1)$ *under* $\mathbb{U}$ *is the connecting map* $\mathbb{M}^\omega 1 \leftarrow \mathbb{M}(\mathbb{M}^\omega 1)$.

Proof. (i) follows from Proposition 6; then (ii) follows from (i) and Proposition 7. Finally (iii) follows from (ii) and Proposition 6. □

Theorem 1. *The greatest fixed point of the functor* $\mathbb{V}(-)^{\mathrm{Act}}$ *can be given the structure of a final labelled Markov process.*

Proof. For simplicity we prove the theorem for the case that Act is a singleton. Since $\mathbb{V}$ restricts to a locally continuous functor on $\omega\mathsf{Coh}$, the fixed point theorem of Smyth and Plotkin [21] tells us that the connecting map $\mathbb{V}^\omega 1 \leftarrow \mathbb{V}(\mathbb{V}^\omega 1)$ is an isomorphism. It follows from Proposition 8 (iii), that the connecting map $\mathbb{M}^\omega 1 \leftarrow \mathbb{M}(\mathbb{M}^\omega 1)$ is also an isomorphism. The inverse of this last map gives $\mathbb{M}^\omega 1$ the structure of an $\mathbb{M}$-coalgebra. That this coalgebra is final follows from a simple categorical argument, cf. [1]. □

Remark 1. Desharnais *et al.* [10] consider the solution of the domain equation $D \cong \mathbb{V}(D)^{\mathrm{Act}}$. Theorem 1 shows that D can be given the structure of a final labelled Markov process. By similar reasoning, D in its Scott topology, can be given the structure of a final coalgebra of the endofunctor $\mathbb{V}(-)^{\mathrm{Act}}$ on Top. We exploit this last observation in Proposition 9.

7 A Metric for the Lawson Topology

Now consider the domain D from Remark 1 qua labelled Markov process; denote the transition probability function by μ. For any formula $f \in \mathcal{F}$, the induced map $f\colon D \to [0,1]$ is monotone and Lawson continuous. This follows by induction on $f \in \mathcal{F}$ using the coincidence of the Lawson and weak topologies on $\mathbb{V}D$. We define a preorder $\preccurlyeq$ on D by $x \preccurlyeq y$ iff $f(x) \leqslant f(y)$ for all $f \in \mathcal{F}$. Since each formula gets interpreted as a monotone function on D it holds that $x \sqsubseteq y$ implies $x \preccurlyeq y$. The theorem below asserts that the converse also holds.

Theorem 2. *The order on* D *coincides with* $\preccurlyeq$.

Desharnais *et al.* [10] have proven a corresponding version of Theorem 2 in which formulas have the usual Boolean semantics. In fact, one can deduce Theorem 2 from this result and another result of the same authors [9, Corollary 3.8] which relates the Boolean and real valued semantics for the logic in the case of finite labelled Markov processes. However, we include a direct topological proof (below) as a nice application of the Lawson = weak coincidence, and because we will need to use this theorem later.

Note that in the following proposition we distinguish between an upper set $V \subseteq D$, and a $\preccurlyeq$-upper set $U \subseteq D$ ($x \in U$ and $x \preccurlyeq y$ implies $y \in U$).

Proposition 9. *If $a \in \mathrm{Act}$, $x \preccurlyeq y$ and $U \subseteq D$ is Scott open and $\preccurlyeq$-upper, then $\mu_{x,a}(U) \leqslant \mu_{y,a}(U)$.*

Proof. Since U is the countable union of sets of the form $\uparrow K$ for finite subsets K of U, it suffices to show that $\mu_{x,a}(\uparrow K) \leq \mu_{y,a}(\uparrow K)$ for all finite subsets K of U.

Let $K = \{x_1, \ldots, x_n\} \subseteq U$ and $z \in D \setminus U$ be given. For each $j \in \{1, ..., n\}$, since $x_j \not\preccurlyeq z$, there exists a formula $g_j \in \mathcal{F}$ such that $g_j(x_j) > g_j(z)$. Since $\mathcal{F}$ is closed under truncated subtraction, and each g_j is Lawson continuous, we may, without loss of generality, assume that $g_j(x_j) > 0$ and g_j is identically zero in a Lawson open neighbourhood of z.

If we set $g = \max_j g_j$, then $g \in \mathcal{F}$ is identically zero in a Lawson open neighbourhood of z and is bounded away from zero on $\uparrow K$. Since $D \setminus U$ is Lawson compact (being Lawson closed) and $\mathcal{F}$ is closed under finite minima, we obtain $f \in \mathcal{F}$ such that f is identically zero on $D \setminus U$ and is bounded away from zero on $\uparrow K$ by, say, $r > 0$. Finally, setting $h = \min(f, r)$, we get

$$\mu_{x,a}(\uparrow K) \leqslant \frac{1}{r} \int h \, d\mu_{x,a} \leqslant \frac{1}{r} \int h \, d\mu_{y,a} \leqslant \mu_{y,a}(U),$$

where the middle inequality follows from $(\langle a \rangle h)(x) \leqslant (\langle a \rangle h)(y)$.

Since U is the (countable) directed union of sets of the form $\uparrow K$ for finite $K \subseteq U$, it follows that $\mu_{x,a}(U) \leqslant \mu_{y,a}(U)$. □

We can now complete the proof of Theorem 2. Let ΣD denote the Scott topology on D and τ the topology of Scott open $\preccurlyeq$-upper sets. Consider the following diagram, where ι is the continuous map given by $\iota x = x$.

$$\begin{array}{ccc} (D, \Sigma D) & \xrightarrow{\mu} & \mathbb{V}(D, \Sigma D)^{\mathrm{Act}} \\ \downarrow{\scriptstyle \iota} & & \downarrow{\scriptstyle \mathbb{V}\iota^{\mathrm{Act}}} \\ (D, \tau) & \dashrightarrow & \mathbb{V}(D, \tau)^{\mathrm{Act}} \end{array} \tag{3}$$

All the solid maps are bijections, so there is a unique dotted arrow making the diagram commute in the category of sets. The inverse image of a sub-basic open in $\mathbb{V}\langle D, \tau \rangle$ under the dotted arrow is τ-open by Proposition 9. By the finality of $\langle D, \mu \rangle$ qua $\mathbb{V}(-)^{\mathrm{Act}}$-coalgebra, ι has a continuous left inverse, and is thus a $\mathbb{V}^{\mathrm{Act}}$-homomorphism. Hence, for each $y \in D$, the Scott closed set $\downarrow y$ is τ-closed, and thus $\preccurlyeq$-lower. Thus $x \preccurlyeq y$ implies $x \sqsubseteq y$. □

Since we view D as a labelled Markov process, we can consider the metric d on D as defined in Section 2.

Theorem 3. *The Lawson topology on D is induced by d.*

Proof. Since the Lawson topology on D is compact, and, by Theorem 2, the topology induced by d is Hausdorff, it suffices to show that the Lawson topology

is finer. Now if $x_n \to x$ in the Lawson topology, then $f(x_n) \to f(x)$ for each $f \in \mathcal{F}$, since each formula gets interpreted as a Lawson continuous map. But d may be uniformly approximated on D to any given tolerance by looking at a finite set of formulas, cf. [6, Lemma 3]. (This lemma crucially uses the assumption $c < 1$ from the definition of d.) Thus $d(x_n, x) \to 0$ as $n \to \infty$. □

8 Testing

Our aim in this section is to characterize the order on the domain D as a testing preorder. The testing formalism we use is that set forth by Larsen and Skou [16]; the idea is to specify an interaction between an experimenter and a process. The way a process responds to the various kinds of tests determines a simple and intuitive behavioural semantics.

Definition 6. *The set of* tests $t \in \mathcal{T}$ *is defined according to the grammar*

$$t ::= \omega \mid a.t \mid (t, \ldots, t),$$

where $a \in \mathrm{Act}$.

The most basic kind of test, denoted ω, does nothing but successfully terminate. $a.t$ specifies the test: see if the process is willing to perform the action a, and in case of success proceed with the test t. Finally, $(t_1, \ldots, t_n)$ specifies the test: make n copies of (the current state of) the process and perform the test t_i on the i-th copy for each i. This facility of copying or replicating processes is crucial in capturing branching-time equivalences like bisimilarity. We usually omit to write ω in non-trivial tests.

Definition 7. *To each test* t *we associate a set* O_t *of* possible observations *as follows.*

$$O_\omega = \{\omega^\surd\} \qquad O_{a.t} = \{a^\times\} \cup \{a^\surd e \mid e \in O_t\} \qquad O_{(t_1,\ldots,t_n)} = O_{t_1} \times \ldots \times O_{t_n}.$$

The only observation of the test ω is successful termination, $\omega^\surd$. Upon performing $a.t$ one possibility, denoted by $a^\times$, is that the a-action fails (and so the test terminates unsuccessfully). Otherwise, the a-action succeeds and we proceed to observe e by running t in the next state; this is denoted $a^\surd e$. Finally an observation of the test $(t_1, ..., t_n)$ is a tuple $(e_1, ..., e_n)$ where each e_i is an observation of t_i.

Definition 8. *For a given test* t*, each state* x *of a labelled Markov process* $\langle X, \Sigma, \mu\rangle$ *induces a probability distribution* $P_{t,x}$ *on* O_t*. The definition of* $P_{t,x}$ *is by structural induction on* t *as follows.*

$$P_{\omega,x}(\omega^\surd) = 1, \quad P_{a.t,x}(a^\times) = 1 - \mu_{a,x}(X), \quad P_{a.t,x}(a^\surd e) = \int (\lambda y.P_{t,y}(e))d\mu_{a,x}$$
$$P_{(t_1,\ldots,t_n),x}(e_1, \ldots, e_n) = \textstyle\prod_{i=1}^n P_{t_i,x}(e_i).$$

The following theorem, proved in an earlier paper [6], shows how bisimilarity may be characterized using the testing framework outlined above. This generalizes a result of Larsen and Skou from discrete probabilistic transition systems satisfying the minimal deviation assumption[4] to labelled Markov processes.

Theorem 4. *Let $\langle X, \Sigma, \mu \rangle$ be a labelled Markov process. Then $x, y \in X$ are bisimilar if and only if $P_{t,x}(E) = P_{t,y}(E)$ for each test t and $E \subseteq O_t$, where $P_{t,x}(E) = \sum_{e \in E} P_{t,x}(e)$.*

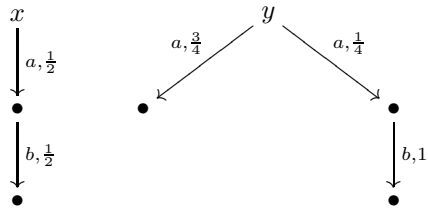
In fact the statement of Theorem 4 can be sharpened somewhat, as we now explain. For each test t there is a distinguished observation, denoted $t^{\surd}$, representing complete success – no action is refused. For instance, if $t = a.(b, c)$ then the completely successful observation is $a^{\surd}(b^{\surd}, c^{\surd})$.

Theorem 5. *Let $\langle X, \Sigma, \mu \rangle$ be a labelled Markov process. Then $x, y \in X$ are bisimilar iff $P_{t,x}(t^{\surd}) = P_{t,y}(t^{\surd})$ for all tests t.*

The idea is that for any test t and $E \subseteq O_t$, the probability of observing E can be expressed in terms of the probabilities of making completely successful observations on all the different 'subtests' of t using the principle of inclusion-exclusion. For example, if $t = a.(b, c)$; then the probability of observing $a^{\surd}(b^{\surd}, c^{\times})$ in state x is equal to $P_{t_1,x}(t_1^{\surd}) - P_{t,x}(t^{\surd})$ where $t_1 = a.b$.

Given Theorem 5 one might conjecture that x is simulated by y if and only if $P_{t,x}(t^{\surd}) \leqslant P_{t,y}(t^{\surd})$ for all tests t. However, the following example shows that to characterize simulation one really needs negative observations.

Example 2. Consider the labelled Markov process $\langle X, \Sigma, \mu \rangle$ depicted below, with distinguished states x and y and label set $\mathrm{Act} = \{a, b\}$.



It is readily verified that $P_{t,x}(t^{\surd}) \leqslant P_{t,y}(t^{\surd})$ for all tests t. However x is not simulated by y. Indeed, consider the test $t = a.(b, b)$ with

$$E = \{a^{\surd}(b^{\times}, b^{\surd}), a^{\surd}(b^{\surd}, b^{\times}), a^{\surd}(b^{\surd}, b^{\surd})\}.$$

If x were simulated by y, then it would follow from Theorem 6 that $P_{t,x}(E) \leqslant P_{t,y}(E)$. But it is easy to calculate that $P_{t,x}(E) = 3/8$ and $P_{t,y}(E) = 1/4$; thus E witnesses the fact that x is not simulated by y.

[4] A discrete labelled Markov process $\langle X, \Sigma, \mu \rangle$ satisfies the minimal deviation assumption if the set $\{ \mu_{x,a}(y) \mid x, y \in X \}$ is finite for each $a \in \mathrm{Act}$.

The example above motivates the following definition. For each test t we define a partial order $\leqslant$ on the set of observations O_t as follows.

1. $a^{\times} \leqslant a^{\surd} e$
2. $a^{\surd} e \leqslant a^{\surd} e'$ if $e \leqslant e'$
3. $(e_1, ..., e_n) \leqslant (e'_1, ..., e'_n)$ if $e_i \leqslant e'_i$ for $i \in \{1, ..., n\}$.

Theorem 6. *Let $\langle X, \Sigma, \mu \rangle$ be a labelled Markov process. Then $x \in X$ is simulated by $y \in X$ iff $P_{t,x}(E) \leqslant P_{t,y}(E)$ for all tests t and upper sets $E \subseteq O_t$.*

We stress that the Theorem characterizes simulation in terms of the measure of *upper sets* E. The 'only if' direction in the above theorem follows from a straightforward induction on tests. The proof of the 'if' direction relies on the definition and lemma below. The idea behind Definition 9 is that one can determine the approximate value of a PML formula in a state x by testing x. This is inspired by [16, Theorem 8.4] where Larsen and Skou show how to determine the truth or falsity of a PML formula using testing. Our approach differs in two respects. Firstly, since we restrict our attention to the positive fragment of the logic it suffices to consider upward closed sets of observations. Also, since we interpret formulas as real-valued functions we can test for the approximate truth value of a formula. It is this last fact that allows us to dispense with the minimal deviation assumption and more generally the assumption of the discreteness of the state space.

Definition 9. *Let $\langle X, \Sigma, \mu \rangle$ be a labelled Markov process. Let $f \in \mathcal{F}$, $0 \leqslant \alpha < \beta \leqslant 1$ and $\delta > 0$. Then there exists a $t \in \mathcal{T}$ and $E \subseteq O_t$ such that for all $x \in X$,*

- *whenever $f(x) \geq \beta$ then $P_{t,x}(E) \geq 1 - \delta$ and*
- *whenever $f(x) \leq \alpha$ then $P_{t,x}(E) \leq \delta$.*

In this case, we say that t is a test for $\langle f, \alpha, \beta \rangle$ with evidence set E and significance level δ.

Thus, if we run t in state x and observe $e \in E$ then with high confidence we can assert that $f(x) > \alpha$. On the other hand, if we observe $e \notin E$ then with high confidence we can assert that $f(x) < \beta$.

Lemma 1. *Let $\langle X, \Sigma, \mu \rangle$ be a labelled Markov process. Then for any $f \in \mathcal{F}$, $0 \leqslant \alpha < \beta \leqslant 1$ and $\delta > 0$, there is a test t for (f, α, β) with level of significance δ and whose associated evidence set $E \subseteq O_t$ is upward closed.*

A proof of Lemma 1 may be found in an appendix to a fuller version of this paper [7]. The lemma implies that if $P_{t,x}(E) \leqslant P_{t,y}(E)$ for all tests t and upper sets $E \subseteq O_t$, then $f(x) \leqslant f(y)$ for all PML formulas f. It follows from Theorem 2 that x is simulated by y. This completes the proof of the 'if' direction of Theorem 6.

9 Summary and Future Work

The theme of this paper has been the use of domain-theoretic and coalgebraic techniques to analyze labelled Markov systems. These systems, which generalize the discrete labelled probabilistic processes investigated by Larsen and Skou [16], have been studied by Desharnais *et al* [8,9,10] and in earlier papers by some of the authors of this paper [4,5,6]. In part, we use domain theory to replace more traditional functional-analytic techniques in earlier papers.

In future, we intend to apply our domain theoretic approach in the more general setting of processes which feature both nondeterministic and probabilistic choice. We believe such a model will be useful in a number of areas, including for example in the analysis of leak rates in covert channels that arise in the study of non-interference.

References

1. Adámek, J. and V. Koubek: On the greatest fixed point of a set functor. Theoretical Computer Science, 150 (1995) 57–75.
2. Alvarez-Manilla, M., A. Edalat, and N. Saheb-Djahromi. An extension result for continuous valuations. Journal of the London Mathematical Society, 61 (2000) 629–640.
3. Averson, W.: An Invitation to C*-Algebras, Springer-Verlag, 1976.
4. van Breugel, F. and J. Worrell: Towards quantitative verification of probabilistic transition system, Lecture Notes in Computer Science, 2076 (2001), Springer-Verlag, 421–432.
5. van Breugel, F. and J. Worrell: An algorithm for quantitative verification of probabilistic transition systems, Lecture Notes in Computer Science, 2154 (2001), Springer-Verlag, 336–350.
6. van Breugel, F., S. Shalit and J. Worrell: Testing labelled Markov processes, Lecture Notes in Computer Science, 2380 (2002), Springer-Verlag, 537–548.
7. van Breugel, F., M. Mislove, J. Ouaknine and J. Worrell: An intrinsic characterization of approximate probabilistic bisimilarity, Report CS-2003-01, York University, 2003.
8. Desharnais, J., A. Edalat and P. Panangaden: A logical characterization of bisimulation for labelled Markov processes, In *Proc. 13th IEEE Symposium on Logic in Computer Science* (1998), IEEE Press, 478–487.
9. Desharnais, J., V. Gupta, R. Jagadeesan and P. Panangaden: Metrics for labeled Markov systems, Lecture Notes in Computer Science, 1664 (1999), Springer-Verlag, 258–273.
10. Desharnais, J., V. Gupta, R. Jagadeesan and P. Panangaden: Approximating labeled Markov processes. In *Proc. 15th Symposium on Logic in Computer Science*, (2000), IEEE Press, 95–106.
11. Edalat, A.: When Scott is weak at the top, Mathematical Structures in Computer Science, 7 (1997), 401–417.
12. Giry, M.: A categorical approach to probability theory, Lecture Notes in Mathematics, 915 (1981), Springer-Verlag, 68–85.
13. Heckmann, R.: Spaces of valuations, Annals of the New York Academy of Sciences, 806 (1996), New York Academy of Sciences, 174–200.

14. Jones, C.: Probabilistic nondeterminism, PhD Thesis, Univ. of Edinburgh, 1990.
15. Jung, A. and R. Tix: The troublesome probabilistic powerdomain. Electronic Notes in Theoretical Computer Science, 13 (1998), Elsevier.
16. Larsen, K.G. and A. Skou: Bisimulation through probabilistic testing: Information and Computation, 94 (1991), Academic Press, 1–28.
17. Milner, R.: Communication and Concurrency, Prentice Hall, 1989.
18. Park, D.: Concurrency and automata on infinite sequences. Lecture Notes in Computer Science, 104 (1981), Springer-Verlag, 167–183.
19. Parthasarathy, K.R.: Probability Measures on Metric Spaces, Academic Press, 1967.
20. Di Pierro, A., C. Hankin, and H. Wiklicky, Approximate non-interference, In *Proc. 15th IEEE Computer Security Foundations Workshop* (2002), IEEE Press, 3-17.
21. Smyth, M.B. and G.D. Plotkin: The category theoretic solution of recursive domain equations, SIAM Journal of Computing, 11 (1982), 761–783.

Manipulating Trees with Hidden Labels

Luca Cardelli - Microsoft Research
Philippa Gardner - Imperial College London
Giorgio Ghelli - Università di Pisa

Abstract. We define an operational semantics and a type system for manipulating semistructured data that contains hidden information. The data model is simple labeled trees with a hiding operator. Data manipulation is based on pattern matching, with types that track the use of hidden labels.

1 Introduction

1.1 Languages for Semistructured Data

XML and semistructured data [1] are inspiring a new generation of programming and query languages based on more flexible type systems [26, 5, 6, 15]. Traditional type systems are grounded on mathematical constructions such as cartesian products, disjoint unions, function spaces, and recursive types. The type systems for semistructured data, in contrast, resemble grammars or logics, with untagged unions, associative products, and Kleene star operators. The theory of formal languages, for strings and trees, provides a wealth of ready results, but it does not account, in particular, for functions. Some integration of the two approaches to type systems is necessary [26, 5].

While investigating semistructured data models and associated languages, we became aware of the need for manipulating private data elements, such as XML identifiers, unique node identifiers in graph models [7], and even heap locations. Such private resources can be modeled using *names* and *name hiding* notions arising from the π-calculus [27]: during data manipulation, the identity of a private name is not important as long as the distinctions between it and other (public or private) names are preserved. Recent progress has been made in handling private resources in programming. FreshML [21] pioneers the *transposition* [30], or swapping, of names, within a type systems that prevents the disclosure of private names.

Other recent techniques can be useful for our purposes. The spatial logics of concurrency devised to cope with π-calculus restriction and scope extrusion [27], and the separation logics used to describe data structures [28,29], provide novel logical operators that can be used also in type systems. Moreover, the notion of dependent types, when the dependence is restricted to names, is tractable [25].

In this paper we bring together a few current threads of development: the effort to devise new languages, type systems, and logics for data structures, the logical operators that come from spatial and nominal logics for private resources, the techniques of transpositions, and the necessity to handle name-dependent types when manipulating private resources. We study these issues in the context of a simplified data model: simple labeled trees with hidden labels, and programs that manipulate such trees. The edges of such trees are labeled with *names*. Our basic techniques can be applied to related data models, such as graphs with hidden node and edge labels, which will be the subject of further work.

1.2 Data Model

The data model we investigate here has the following constructors. Essentially, we extend a simple tree model (such as XML) in a general and orthogonal way with a hiding operator.

0	the tree consisting of a single root node;
$n[P]$	the tree with a single edge from the root, labeled n, leading to P;

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 216-230, 2003.

$P \mid Q$	the root-merge of two trees (commutative and associative);
$(\nu n)P$	the tree P where the label n is hidden/private/restricted. As in π-calculus, we call *restriction* the act of hiding a name.

Trees are inspected by pattern matching. For example, program (1) below inspects a tree t having shape $n[P] \mid Q$, for some P,Q, and produces $P \mid m[Q]$. Here n,m are constant (public) labels, x,y are pattern variables, and $\mathbf{T}$ is both the pattern that matches any tree and the type of all trees. It is easy to imagine that, when parameterized in t, this program should have the type indicated.

$$\textbf{\textit{match}}\ t\ \textbf{\textit{as}}\ (n[x{:}\mathbf{T}] \mid y{:}\mathbf{T})\ \textbf{\textit{then}}\ (x \mid m[y]\ ) \qquad (1)$$
transforms a tree $t = n[P] \mid Q$ into $P \mid m[Q]$
expected typing: $(n[\mathbf{T}] \mid \mathbf{T}) \to (\mathbf{T} \mid m[\mathbf{T}])$

Using the same pattern match as in (1), let us now remove the public label n and insert a private one, p, that is created and bound to the program variable z at "run-time":

$$\textbf{\textit{match}}\ t\ \textbf{\textit{as}}\ (n[x{:}\mathbf{T}] \mid y{:}\mathbf{T})\ \textbf{\textit{then}}\ (\nu z)\ (x \mid z[y]) \qquad (2)$$
transforms $t = n[P] \mid Q$ into $(\nu p)\ (P \mid p[Q])$ for a fresh label p
expected typing: $(n[\mathbf{T}] \mid \mathbf{T}) \to (\mathsf{H}z.\ (\mathbf{T} \mid z[\mathbf{T}]))$

In our type system, the *hidden name quantifier* H is the type construct corresponding to the data construct ν [10]. More precisely, $\mathsf{H}z.\mathcal{A}$ means that there is a hidden label p denoted by the variable z, such that the data is described by $\mathcal{A}\{z{\leftarrow}p\}$. (Scope extrusion [27] makes the relationship non trivial, see Sections 2 and 4.) Because of the $\mathsf{H}z.\mathcal{A}$ construct, types contain name variables; that is, types are dependent on names.

The first two examples pattern match on the public name n. Suppose instead that we want to find and manipulate private names. The following example is similar to (2), except that now a private label p from the data is matched and bound to the variable z.

$$\textbf{\textit{match}}\ t\ \textbf{\textit{as}}\ ((\nu z)\ (z[x{:}\mathbf{T}] \mid y{:}\mathbf{T}))\ \textbf{\textit{then}}\ x \mid z[y] \qquad (3)$$
transforms $t = (\nu p)(p[P] \mid Q)$ into $(\nu p)(P \mid p[Q])$
expected typing: $(\mathsf{H}z.\ (z[\mathbf{T}] \mid \mathbf{T})) \to (\mathsf{H}z.\ (\mathbf{T} \mid z[\mathbf{T}]))$

Note that the restriction (νp) in the result is not apparent in the program: it is implicitly applied by a match that opens a restriction, so that the restricted name does not escape.

As the fourth and remaining case, we convert a private name in the data into a public one. The only change from (3) is a public name m instead of z in the result:

$$\textbf{\textit{match}}\ t\ \textbf{\textit{as}}\ ((\nu z)\ (z[x{:}\mathbf{T}] \mid y{:}\mathbf{T}))\ \textbf{\textit{then}}\ x \mid m[y] \qquad (4)$$
transforms $t = (\nu p)\ (p[P] \mid Q)$ into $(\nu p)(P \mid m[Q])$
expected typing: $(\mathsf{H}z.\ (z[\mathbf{T}] \mid \mathbf{T})) \to (\mathsf{H}z.\ (\mathbf{T} \mid m[\mathbf{T}]))$

This program replaces only one occurrence of p: the residual restriction (νp) guarantees that any other occurrences inside P,Q remain bound. As a consequence, the binder $\mathsf{H}z$ has to remain in the result type. Note that, although we can replace a private name with a public one, we cannot "expose" a private name, because of the rebinding of the output.

As an example of an incorrectly typed program consider the following attempt to assign a simpler type to the result of example (4), via a typed ***let*** biding:

$$\textbf{\textit{let}}\ w : (\mathbf{T} \mid m[\mathbf{T}]) = \textbf{\textit{match}}\ t\ \textbf{\textit{as}}\ ((\nu z)\ (z[x{:}\mathbf{T}] \mid y{:}\mathbf{T}))\ \textbf{\textit{then}}\ x \mid m[y]$$

Here we would have to check that $\mathsf{H}z.\ (\mathbf{T} \mid m[\mathbf{T}])$ is compatible with $(\mathbf{T} \mid m[\mathbf{T}])$. This would work if we could first show that $\mathsf{H}z.\ (\mathbf{T} \mid m[\mathbf{T}])$ is a subtype of $(\mathsf{H}z.\ \mathbf{T}) \mid (\mathsf{H}z.\ m[\mathbf{T}])$, and then simplify. But such a subtyping does not hold since, e.g., $(\nu p)(p[0] \mid m[p[0]])$ matches the former type but not the latter, because the restriction (νp) cannot be distributed.

So far, we have illustrated the manipulation of individual private or public names by pattern matching and data constructors. However, we may want to replace throughout a whole data structure a public name with another, or a public one with a private one, or vice versa. We could do this by recursive analysis, but it would be very difficult to reflect what has happened in the type structure, likely resulting in programs of type $\mathbf{T}\rightarrow\mathbf{T}$. So, we introduce a *transposition* facility as a primitive operation, and as a corresponding type operator. In the simplest case, if we want to transpose (exchange) a name n with a name m in a data structure t we write $t(n\leftrightarrow m)$. If t has type $\mathcal{A}$, then $t(n\leftrightarrow m)$ has type $\mathcal{A}(n\leftrightarrow m)$. We define rules to manipulate type level transpositions; for example we derive that, as types, $n[\mathbf{0}](n\leftrightarrow m) = m[\mathbf{0}]$.

Transposition types are interesting when exchanging public and private labels. Consider the following program and its initial syntax-driven type:

$$\lambda x{:}n[\mathbf{T}].\ (\nu z)\ x(m\leftrightarrow z)\quad :\quad n[\mathbf{T}]\rightarrow \mathsf{H}z.n[\mathbf{T}](m\leftrightarrow z)\qquad (=\ n[\mathbf{T}]\rightarrow n[\mathbf{T}]\ ) \tag{5}$$

This program takes data of the form $n[P]$, creates a fresh label p denoted by z, and swaps the public m with the fresh p in $n[P]$, to yield $(\nu p)n[P](m\leftrightarrow p)$, where the fresh p has been hidden in the result. Since n is a constant different from m, and p is fresh, the result is in fact $(\nu p)n[P(m\leftrightarrow p)]$. The result type can be similarly simplified to $\mathsf{H}z.n[\mathbf{T}(m\leftrightarrow z)]$. Now, swapping two names in the set of all trees, $\mathbf{T}$, produces again the set of all trees. Therefore, the result type can be further simplified to $\mathsf{H}z.n[\mathbf{T}]$. We then have that $\mathsf{H}z.n[\mathbf{T}] = n[\mathbf{T}]$, since a restriction can be pushed through a public label, where it is absorbed by $\mathbf{T}$. Therefore, the type of our program is $n[\mathbf{T}]\rightarrow n[\mathbf{T}]$.

Since we already need to handle name-dependent types, we can introduce, without much additional complexity, a dependent function type $\Pi w.\ \mathcal{A}$. This is the type of functions $\lambda w{:}\mathbf{N}.t$ that take a name m (of type $\mathbf{N}$) as input, and return a result of type $\mathcal{A}\{w\leftarrow m\}$. We can then write a more parametric version of example (5), where the constant n is replaced by a name variable w which is a parameter:

$$\lambda w{:}\mathbf{N}.\ \lambda x{:}w[\mathbf{T}].\ (\nu z)\ x(m\leftrightarrow z)\quad :\quad \Pi w.\ (w[\mathbf{T}]\rightarrow \mathsf{H}z.\ w[\mathbf{T}](m\leftrightarrow z)\ ) \tag{6}$$

Now, the type $\mathsf{H}z.\ w[\mathbf{T}](m\leftrightarrow z)$ simplifies to $\mathsf{H}z.\ w(m\leftrightarrow z)[\mathbf{T}]$, but no further, since m can in fact be given for w, in which case it would be transposed to the private z.

Transpositions are emerging as a unifying and simplifying principle in the formal manipulation of binding operators [30], which is a main goal of this paper. If some type-level manipulation of names is of use, then transpositions seem a good starting point.

1.3 Related and Future Work

It should be clear from Section 1.2 that sophisticated type-level manipulations are required for our data model, involving transposition types (which seem to be unique to our work), hiding quantifiers, and dependent types. Furthermore, we work in the context of a data model and type system that is "non-structural", both in the sense of supporting grammar-like types (with $\wedge\ \vee\ \neg$) and in the sense of supporting π-calculus-style extruding scopes. In both these aspects we differ from FreshML [31], although we base much of our development on the same foundations [30]. Our technique of automatically rebinding restrictions sidesteps some complex issues in the FreshML type system, and yet seems to be practical for many examples. FreshML uses "apartness types" $\mathcal{A}\#w$, which can be used to say that a function takes a name denoted by w and a piece of data $\mathcal{A}$ that does not contain that name. We can express that idiom differently as $\Pi w.\ (\mathcal{A}\wedge\neg\copyright w)\rightarrow\mathcal{B}$, where the operator $\copyright w$ [10] means "contains free the name denoted by w".

Our calculus is based on a pattern matching construct that performs run-time type tests; in this respect, it is similar to the XML manipulation languages XDuce [26] and CDuce [5]. However, those languages do not deal with hidden names, whose study is our main goal.

XDuce types are based on tree grammars: they are more restrictive than ours but are based on well-known algorithms. CDuce types are in some aspects richer than ours: they mix the logical and functional levels that we keep separate; such mixing would not easily extend to our $\mathsf{H}x.\mathcal{A}$ types. Other differences stem from the data model (our $P \mid Q$ is commutative), and from auxiliary programming constructs.

The database community has defined many languages to query semistructured data [1,3,6,8,15,17,19,20], but they do not deal with hidden names. The theme of hidden identifiers (OIDs) has been central in the field of object-oriented database languages [2, 4]. However, the debate there was between languages where OIDs are hidden to the user, and lower-level languages where OIDs are fully visible. The second approach is more expressive but has the severe problem that OIDs lose their meaning once they are exported outside their natural scope. We are not aware of any proposal with operators to define a scope for, reveal, and rehide private identifiers, as we do in our calculus.

In TQL [15], the semistructured query language closest to this work, a programmer writes a logical formula, and the system chooses a way to retrieve all pieces of data that satisfy that formula. In our calculus, such formulas are our tree types, but the programmer has to write the recursion patterns that collect the result (as in Section 8). The TQL approach is best suited to collecting the subtrees that satisfy a condition, but the approach we explore here is much more expressive; for example, we can apply transformations at an arbitrary depth, which is not possible in TQL. Other query-oriented languages, such as XQuery [6], support structural recursion as well, for expressiveness.

As a major area of needed future work, our subtyping relation is not prescribed in detail here (apart for the non-trivial subtypings coming from transposition equivalence). Our type system is parameterized by an unspecified set of ValidEntailments, which are simply assumed to be sound for typing purposes. The study of related subtyping relations (a.k.a. valid logical implications in spatial logics [11]) is in rapid development. The work in [12] provides a complete subtyping algorithm for ground types (i.e. not including $\mathsf{H}z.\mathcal{A}$), and other algorithms are being developed that include Kleene star [18]. Such theories and algorithms could be taken as the core of our ValidEntailments. But adding quantifiers is likely to lead to either undecidability or incompleteness. In the middle ground, there is a collection of sound and practical inclusion rules [10,11] that can be usefully added to the ground subtyping relation (e.g., $\mathsf{H}z.n[\mathcal{A}] <: n[\mathsf{H}z.\mathcal{A}]$ for example (5)). By parameterizing over the ValidEntailments, we show that these issues are relatively orthogonal to the handling of transpositions and hiding.

A precursor of this work handles a simpler data model, with no hiding but with a similarly rich type system based on spatial logic [12]. However, even the richer data model considered in this paper is not all one could wish for. For example, hiding makes better sense for graph nodes [14], or for addresses in heaps [29], than for tree labels. More sophisticated data models include graphs, and combinations of trees and graphical links as in practical uses of XML (see example in Section 8). In any case, the manipulation of hidden resources in data structures is fundamental.

2 Values

Our programs manipulate *values*; either *name values* (from a countable set of names Λ), *tree values*, or *function values* (i.e., closures). Over the tree values, we define a structural congruence relation $\equiv$ that factors out the equivalence laws for $\mid$ and 0, and the scoping laws for restriction. Function values are triples of a term t (Section 3) with respect to an input variable x (essentially, $\lambda x.t$) and a stack for free variables ρ. A stack ρ is a list of bindings of variables to values. Name transpositions are defined on all values.

2-1 Definition: Tree Values

Λ	Names: a countable set of names *n, m, p*, ...		
$P,Q,R ::=$	Tree values	All names: $na(P)$	Free names: $fn(P)$
0	void	$na(0) \triangleq \{\}$	$fn(0) \triangleq \{\}$
$P \mid Q$	composition	$na(P \mid Q) \triangleq na(P) \cup na(Q)$	$fn(P \mid Q) \triangleq fn(P) \cup fn(Q)$
$n[P]$	location	$na(n[P]) \triangleq \{n\} \cup na(P)$	$fn(n[P]) \triangleq \{n\} \cup fn(P)$
$(\nu n)P$	restriction	$na((\nu n)P) \triangleq \{n\} \cup na(P)$	$fn((\nu n)P) \triangleq fn(P) - \{n\}$

We define an *actual transposition* operation on tree values, $P\bullet(m\leftrightarrow m')$, that blindly swaps free and bound names *m,m'* within *P*. The interaction of transpositions with binders such as $(\nu n)P$ supports a general formal treatment of bound names [30].

2-2 Definition: Actual Transposition of Names and Tree Values.

$n\bullet(m\leftrightarrow m) = m$ $\qquad$ $0\bullet(m\leftrightarrow m') = 0$

$n\bullet(m\leftrightarrow n) = m$ $\qquad$ $(P \mid Q)\bullet(m\leftrightarrow m') = P\bullet(m\leftrightarrow m') \mid Q\bullet(m\leftrightarrow m')$

$n\bullet(m\leftrightarrow m') = n$ if $n\neq m$ and $n\neq m'$ $\qquad$ $n[P]\bullet(m\leftrightarrow m') = n\bullet(m\leftrightarrow m')[P\bullet(m\leftrightarrow m')]$

$((\nu n)P)\bullet(m\leftrightarrow m') = (\nu n\bullet(m\leftrightarrow m'))P\bullet(m\leftrightarrow m')$

Transpositions are used in the definition of α-congruence and capture-avoiding substitution. Structural congruence is analogous to the standard definition for π-calculus [27]; the "scope extrusion" rule for ν over -|- is written in an equivalent equational style.

2-3 Definition: α-Congruence and Structural Congruence on Tree Values.

α-congruence, $\equiv_\alpha$, is the least congruence relation on tree values such that:

$(\nu n)P \equiv_\alpha (\nu m)(P\bullet(n\leftrightarrow m))$ where $m\notin na(P)$

Structural congruence, $\equiv$, is the least congruence relation on tree values such that:

$P \equiv_\alpha Q \;!\; P \equiv Q$ $\qquad$ $(\nu n)0 \equiv 0$

$P \mid Q \equiv Q \mid P$ $\qquad$ $(\nu n)m[P] \equiv m[(\nu n)P]$ if $n\neq m$

$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$ $\qquad$ $(\nu n)(P \mid (\nu n)Q) \equiv ((\nu n)P) \mid ((\nu n)Q)$

$P \mid 0 \equiv P$ $\qquad$ $(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$

N.B.: This notion of α-congruence can be shown equivalent to the standard one.

2-4 Definition: Free Name Substitution on Tree Values.

$0\{n\leftarrow m\} = 0$ $\qquad$ $(P \mid Q)\{n\leftarrow m\} = P\{n\leftarrow m\} \mid Q\{n\leftarrow m\}$

$p[P]\{n\leftarrow m\} = p\{n\leftarrow m\}[P\{n\leftarrow m\}]$

$((\nu p)P)\{n\leftarrow m\} = (\nu q)((P\bullet(p\leftrightarrow q))\{n\leftarrow m\})$ for $q\notin na((\nu p)P)\cup\{n,m\}$

N.B.: different choices of *q* in the last clause, lead to α-congruent results.

We next define high values and transpositions over them (see also Definition 3-1 for the syntax of terms *t*). A stack ρ is a list of pairs of the form $\emptyset[x_1\leftarrow F_1]...[x_n\leftarrow F_n]$, where x_i are variables (distinct from names), F_i are high values, $\rho(x_i) \triangleq F_j$ where *j* is the largest index such that $x_i = x_j$, and $dom(\rho) \triangleq \{x_1, ..., x_n\}$. Variables are not affected by transpositions.

$\langle\rho, x, t\rangle\bullet(n\leftrightarrow n') \triangleq \langle\rho\bullet(n\leftrightarrow n'), x, t\bullet(n\leftrightarrow n')\rangle$

$\emptyset\bullet(n\leftrightarrow n') \triangleq \emptyset$

$\rho[x\leftarrow F]\bullet(n\leftrightarrow n') \triangleq \rho\bullet(n\leftrightarrow n')[x\leftarrow F\bullet(n\leftrightarrow n')]$

2-5 Definition: High Values and Stacks

$F,G,H ::=$	High Values	All names: $na(F)$	Free names: $fn(F)$
n	name values	$na(n) \triangleq \{n\}$	$fn(n) \triangleq \{n\}$
P	tree values	$na(P)$: see tree values	$fn(P)$: see tree values
$\langle\rho, x, t\rangle$	function values	$na(\langle\rho, x, t\rangle) \triangleq na(t)\cup na(\rho)$	$fn(\langle\rho, x, t\rangle) \triangleq fn(t)\cup fn(\rho)$
		$na(\rho) \triangleq \bigcup_{x\in dom(\rho)} na(\rho(x))$	$fn(\rho) \triangleq \bigcup_{x\in dom(\rho)} fn(\rho(x))$

3 Syntax

Our λ-calculus is stratified in terms of *low types* and *high types*. The low types are the tree types and the type of names, **N**. (Basic data types such as integers could be added to low types.) The novel aspects of the type structure are the richness of the tree types, which come from the formulas of spatial logics [16, 10], and the presence of transposition types. We then have higher types over the low types: function types and name-dependent types. The precise meaning of types is given in Section 4.

The same stratification holds on terms, which can be of low or high type, as is more apparent in the operational semantics of Section 5 and in the type rules of Section 7.

3-1 Definition: Syntax

$\mathcal{N},\mathcal{M},\mathcal{P},\mathcal{Q}$ ::=	*Name Expressions*
$x,\ n,\ \mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}$	name variable, name constant, name transposition
$\mathcal{A},\mathcal{B}$ ::=	*Tree Types*
$\mathbf{0},\ \mathcal{N}[\mathcal{A}],\ \mathcal{A}\mid\mathcal{B},\ \mathsf{H}\underline{x}.\mathcal{A},\ ©\mathcal{N},$	void, location, composition, hiding, occurrence,
$\mathbf{F},\ \mathcal{A}\wedge\mathcal{B},\ \mathcal{A}!\ \mathcal{B},\ \mathcal{A}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}$	false, conjunction, implication, type transposition
$\mathcal{F},\mathcal{G},\mathcal{H}$ ::=	*High Types*
$\mathcal{A},\ \mathbf{N},$	tree types, name type,
$\mathcal{F}\to\mathcal{G},\ \Pi\underline{x}.\mathcal{G}$	function types ($\mathcal{F}\neq\mathbf{N}$), dependent types ($x$:**N**)
t,u,v ::=	*Terms*
$0,\ \mathcal{N}[u],\ t\mid u,\ (\nu\underline{x})t,$	void, location, composition, restriction,
$t\div(\mathcal{N}[\underline{y}:\mathcal{A}]).u,\ t\div(\underline{x}:\mathcal{A}\mid\underline{y}:\mathcal{B}).u,$	location match, composition match,
$t\div((\nu\underline{x})\underline{y}:\mathcal{A}).u,\ t?(\underline{x}:\mathcal{A}).u,v,$	restriction match, tree type test,
$t\{\mathcal{M}\leftrightarrow\mathcal{M}'\},$	term transposition,
$x,\ \mathcal{N},\ \lambda\underline{x}:\mathcal{F}.t,\ t(u)$	high variable, name expr, function, application

Underlined variables indicate binding occurrences. The scoping rules should be clear, except that: in location match $\underline{y}$ scopes u; in composition match $\underline{x}$ and $\underline{y}$ scope u; in restriction match $\underline{x}$ scopes $\mathcal{A}$ and u, and $\underline{y}$ scopes u; in tree type test $\underline{x}$ scopes u and v. We define name sets, such as $na(\mathcal{A})$, and actual transpositions on all syntax, such as $t\bullet\{n\leftrightarrow m\}$, in the obvious way (there are no name binders in the syntax). We also define free-variable sets $fv(\text{-})$ on all syntax (based on the mentioned binding occurrences), and capture-avoiding substitutions of name expressions for variables: $\mathcal{N}\{x\leftarrow\mathcal{M}\}$, $\mathcal{A}\{x\leftarrow\mathcal{M}\}$, and $\mathcal{F}\{x\leftarrow\mathcal{M}\}$.

Name expressions, tree types, and terms include *(formal) transposition* operations that are part of the syntax; they represent (actual) transpositions on data, indicated by the • symbol.

The tree types are formulas in a spatial logic, so we can derive the standard types (formulas) for negation $\neg\mathcal{A}\triangleq\mathcal{A}!\ \mathbf{F}$ and disjunction $\mathcal{A}\vee\mathcal{B}\triangleq\neg(\neg\mathcal{A}\wedge\neg\mathcal{B})$.

The terms include a standard λ-calculus fragment, the basic tree constructors, and some matching operators for analyzing tree data. The *tree type test* construct (distinguished by the character '?') performs a run-time check to see whether a tree has a given type: if tree t satisfies type $\mathcal{A}$ then u is run with x of type $\mathcal{A}$ bound to t; otherwise v is run with x of type $\neg\mathcal{A}$ bound to t. In addition, one needs matching constructs (distinguished by the character '÷') to decompose the tree: *composition match* splits a tree in two components, *location match* strips an edge from a tree, and *restriction match* inspects a hidden label in a tree. A *zero match* is redundant because of the tree type test construct. These multiple matching constructs are designed to simplify the operational semantics and the type rules. In practice, one would use a single case statement with patterns over the structure of trees, but this can be encoded.

In the quantifier $\mathsf{H}x.\mathcal{A}$ and in the restriction match construct, the type $\mathcal{A}$ is dependent on

variable x (denoting a hidden name). This induces the need for handling dependent types, and motivates the $\Pi x.\mathcal{G}$ dependent function type constructor. The type dependencies, however, are restricted to name variables, which may be replaced only by name expressions (that is, not by general computations on names). Because of this, these dependent types are relatively easy to handle.

4 Satisfaction

The satisfaction relation, written $\models$, relates values to types, and thus provides the semantic meaning of typing that is enforced by the type system of Section 7. For type constructs such as $\wedge$ and $!$, this is related to the notion of satisfaction from modal logic. Over tree types we have essentially the relation studied in [16, 10], extended to hiding and transpositions. Satisfaction is then generalized to high types, where it depends on the operational semantics $\forall_\rho$ of Section 5, which depends on satisfaction at tree types only.

4-1 Definition: Satisfaction

On Name Expressions: $n \models_N \mathcal{N}$, for $\mathcal{N}$ closed (no free variables), is defined by:

$n \models_N m$	iff $m = n$
$n \models_N \mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}$	iff $\exists m,m'.\ m \models_N \mathcal{M}$ and $m' \models_N \mathcal{M}'$ and $n\bullet(m\leftrightarrow m') \models_N \mathcal{N}$

On Tree Types: $P \models_T \mathcal{A}$, for $\mathcal{A}$ closed, is defined by:

$P \models_T \mathbf{0}$	iff $P \equiv 0$
$P \models_T \mathcal{N}[\mathcal{A}]$	iff $\exists n,P'.\ n \models_N \mathcal{N}$ and $P \equiv n[P']$ and $P' \models_T \mathcal{A}$
$P \models_T \mathcal{A} \mid \mathcal{B}$	iff $\exists P',P''.\ P \equiv P' \mid P''$ and $P' \models_T \mathcal{A}$ and $P'' \models_T \mathcal{B}$
$P \models_T \mathsf{H}x.\mathcal{A}$	iff $\exists n,P'.\ P \equiv (\nu n)P'$ and $n \notin na(\mathcal{A})$ and $P' \models_T \mathcal{A}\{x\leftarrow n\}$
$P \models_T \copyright\mathcal{N}$	iff $\exists n.\ n \models_N \mathcal{N}$ and $n \in fn(P)$
$P \models_T \mathbf{F}$	never
$P \models_T \mathcal{A} \wedge \mathcal{B}$	iff $P \models_T \mathcal{A}$ and $P \models_T \mathcal{B}$
$P \models_T \mathcal{A}\ !\ \mathcal{B}$	iff $P \models_T \mathcal{A}$ implies $P \models_T \mathcal{B}$
$P \models_T \mathcal{A}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}$	iff $\exists m,m'.\ m \models_N \mathcal{M}$ and $m' \models_N \mathcal{M}'$ and $P\bullet(m\leftrightarrow m') \models_T \mathcal{A}$

On High Types: $F \models_H \mathcal{F}$, for $\mathcal{F}$ closed, is defined by:

$F \models_H \mathbf{N}$	iff $F \models_N \mathcal{N}$ for some $\mathcal{N}$
$F \models_H \mathcal{A}$	iff $F \models_T \mathcal{A}$
$H \models_H \mathcal{F}\rightarrow\mathcal{G}$ $(\mathcal{F}\neq\mathbf{N})$	iff $H = \langle \rho, z, t\rangle$ and $\forall F,G.\ (F \models_H \mathcal{F} \wedge\ t\ \forall_{\rho[z\leftarrow F]}\ G)\ !\quad G \models_H \mathcal{G}$
$H \models_H \Pi x.\ \mathcal{G}$	iff $H = \langle \rho, z, t\rangle$ and $\forall n,G.\ t\ \forall_{\rho[z\leftarrow n]}\ G\ !\quad G \models_H \mathcal{G}\{x\leftarrow n\}$

(We will omit the subscripts on $\models$.) The constructs $\mathsf{H}x.\mathcal{A}$ and $\copyright\mathcal{N}$ are derived operators in [10], and are taken here as primitive, in the original spirit of [9]. In the definition of $\mathsf{H}x.\mathcal{A}$, the clause $P\equiv(\nu n)P'$ pulls a restriction (even a dummy one) from elsewhere in the data, via scope extrusion (Definition 2-3). The type $\mathsf{H}w.\copyright w$ is the type of non-redundant restrictions, with the quantifier $\mathsf{H}w$ revealing a restricted name n, and $\copyright w$ declaring that this n is used in the data. The meaning of formal transpositions relies on actual transpositions. At high types, a closure $\langle \rho, z, t\rangle$ satisfies a function type $\mathcal{F}\rightarrow\mathcal{G}$ if, on any input satisfying $\mathcal{F}$, every output satisfies $\mathcal{G}$; similarly for $\Pi x.\ \mathcal{G}$.

4-2 Proposition: Tree Satisfaction Under Structural Congruence.
If $P \models \mathcal{A}$ and $P \equiv Q$ then $Q \models \mathcal{A}$.

4-3 Lemma: Name and Tree Satisfaction Under Actual Transposition.
If $n \models \mathcal{N}$ then $n\bullet(m\leftrightarrow m') \models \mathcal{N}\bullet(m\leftrightarrow m')$. If $P \models \mathcal{A}$ then $P\bullet(m\leftrightarrow m') \models \mathcal{A}\bullet(m\leftrightarrow m')$.

5 Operational Semantics

We give a *big step* operational semantics that is later used for a subject reduction result (Theorem 7-4). This style of semantics, namely a relation between a program and all its potential final results, is sufficient to clarify the intended behavior of our operations. It could be extended with error handling. Alternatively, a *small step* semantics could be given. In either case, one could go further and establish a type soundness theorem stating that well-typed programs (preserve types and) do not get stuck. All this is relatively routine, and we opt to give only the essential semantics.

The operational semantics is given by a relation $t \forall_\rho F$ between terms t, stacks ρ, and values F, meaning that t can evaluate to F on stack ρ. An auxiliary relation, $\mathcal{N} \downarrow_\rho n$, deals with evaluation of name expressions. The semantics of run-time tests makes use of the satisfaction relation from Section 4. We use, $t \forall_\rho P$ to indicate that t evaluates to a tree value. We use $t \forall_\rho \equiv P$ as an abbreviation for $t \forall_\rho Q$ and $Q \equiv P$, for some Q.

5-1 Definition: Operational Semantics

$$\frac{x \in dom(\rho) \quad \rho(x) \in \Lambda}{x \downarrow_\rho \rho(x)} \text{ (NRed } x\text{)} \qquad \frac{}{n \downarrow_\rho n} \text{ (NRed } n\text{)} \qquad \frac{\mathcal{N} \downarrow_\rho n \quad \mathcal{M} \downarrow_\rho m \quad \mathcal{M}' \downarrow_\rho m'}{\mathcal{N}\{\mathcal{M} \leftrightarrow \mathcal{M}'\} \downarrow_\rho n\bullet(m \leftrightarrow m')} \text{ (NRed } \leftrightarrow\text{)}$$

$$\frac{}{0 \forall_\rho 0} \text{ (Red 0)} \qquad \frac{\mathcal{N} \downarrow_\rho n \quad t \forall_\rho P}{\mathcal{N}[t] \forall_\rho n[P]} \text{ (Red } \mathcal{N}[]\text{)}$$

$$\frac{t \forall_\rho P \quad u \forall_\rho Q}{t \mid u \forall_\rho P \mid Q} \text{ (Red } \mid\text{)} \qquad \frac{n \notin na(t, \rho) \quad t \forall_{\rho[x \leftarrow n]} P}{(\nu x)t \forall_\rho (\nu n)P} \text{ (Red } \nu\text{)}$$

$$\frac{t \forall_\rho P \quad \mathcal{M} \downarrow_\rho m \quad \mathcal{M}' \downarrow_\rho m'}{t\{\mathcal{M} \leftrightarrow \mathcal{M}'\} \forall_\rho P\bullet(m \leftrightarrow m')} \text{ (Red } \leftrightarrow\text{)}$$

$$\frac{\mathcal{N} \downarrow_\rho n \quad t \forall_\rho \equiv n[P] \quad P \vDash \rho(\mathcal{A}) \quad u \forall_{\rho[y \leftarrow P]} F}{t \div (\mathcal{N}[y{:}\mathcal{A}]).u \forall_\rho F} \text{ (Red } \div\mathcal{N}[]\text{)}$$

$$\frac{t \forall_\rho \equiv P' \mid P'' \quad P' \vDash \rho(\mathcal{A}) \quad P'' \vDash \rho(\mathcal{B}) \quad x \neq y \quad u \forall_{\rho[x \leftarrow P'][y \leftarrow P'']} F}{t \div (x{:}\mathcal{A} \mid y{:}\mathcal{B}).u \forall_\rho F} \text{ (Red } \div\mid\text{)}$$

$$\frac{n \notin na(t, \mathcal{A}, u, \rho) \quad t \forall_\rho \equiv (\nu n)P \quad P \vDash \rho[x \leftarrow n](\mathcal{A}) \quad x \neq y \quad u \forall_{\rho[x \leftarrow n][y \leftarrow P]} Q}{t \div ((\nu x)y{:}\mathcal{A}).u \forall_\rho (\nu n)Q} \text{ (Red } \div\nu\text{)}$$

$$\frac{t \forall_\rho P \quad P \vDash \rho(\mathcal{A}) \quad u \forall_{\rho[x \leftarrow P]} F}{t ? (x{:}\mathcal{A}).u, v \forall_\rho F} \text{ (Red } ?\vDash\text{)} \qquad \frac{t \forall_\rho P \quad P \vDash \neg\rho(\mathcal{A}) \quad v \forall_{\rho[x \leftarrow P]} F}{t ? (x{:}\mathcal{A}).u, v \forall_\rho F} \text{ (Red } ?\nvDash\text{)}$$

$$\frac{x \in dom(\rho)}{x \forall_\rho \rho(x)} \text{ (Red } x\text{)} \qquad \frac{\mathcal{N} \downarrow_\rho n}{\mathcal{N} \forall_\rho n} \text{ (Red } \mathcal{N}\text{)} \qquad \frac{}{\lambda x{:}\mathcal{T}.t \forall_\rho \langle \rho, x, t \rangle} \text{ (Red } \lambda\text{)} \qquad \frac{t \forall_\rho \langle \rho', x, t' \rangle \quad u \forall_\rho G \quad t' \forall_{\rho'[x \leftarrow G]} H}{t(u) \forall_\rho H} \text{ (Red App)}$$

The operations (Red ÷ -) and (Red ? -) can execute run-time type tests on dependent types that are run-time instantiated; e.g., note the role of x in $\lambda x{:}\mathbf{N}.\ t?(y{:}x[\mathbf{0}]).u, v$. Here, $\rho(\mathcal{A})$ replaces every free variable $x \in dom(\rho)$ in $\mathcal{A}$ with $\rho(x)$. The rules are applicable only if $\rho(\mathcal{A})$ is a well-formed type: the type rules of Section 7 guarantee this condition.

The matching reductions are nondeterministic and, in a big step semantics, avoid divergent paths if convergent paths are possible.

Reduction is not closed up to ≡ (0 does not reduce to 0|0), nor up to $\equiv_\alpha$ (see (Red ν) and (Red ÷ν), which exclude some of the bound names that can be returned). But this is a matter of choice that has no effect on our results.

The following lemma is crucial in the subject reduction cases for (Red ν) (Theorem 7-4). Only this transposition lemma is needed there, not a harder substitution lemma.

5-2 Lemma: Reduction Under Transposition.

If $\mathcal{M} \downarrow_\rho m$ then $\mathcal{M}\bullet(n\leftrightarrow n') \downarrow_{\rho\bullet(n\leftrightarrow n')} m\bullet(n\leftrightarrow n')$.

If $t \forall_\rho F$ then $t\bullet(n\leftrightarrow n') \forall_{\rho\bullet(n\leftrightarrow n')} F\bullet(n\leftrightarrow n')$.

6 Transposition Equivalence and Apartness

We define a type equivalence relation on name expressions and tree types, which in particular allows any type transposition to be eliminated or pushed down to the name expressions that appear in the type. The main aim of this section is to establish the soundness of such an equivalence relation, which is inspired by [22,11]. A crucial equivalence rule, (EqN ↔ Apart), requires the notion of *apartness* of name expressions, meaning that the names that those expressions denote are distinct. (*C.f.* examples (5) and (6) in Introduction.) Apartness of name expressions depends on apartness of variables and names; we keep track of such relationships via a *freshness signature*.

6-1 Definition: Freshness Signature

A *freshness signature* ϕ is an ordered list of distinct variables annotated with either ∀ or H, and of names. (For example: $p,\forall x,\mathsf{H}y,n,m,p,\mathsf{H}z,\forall w$.) Notation: $dom(\phi)$ is the set of variables in ϕ; $na(\phi)$ is the set of names in ϕ; $\phi(x)$ is the symbol ∀ or H associated to x in ϕ. We write $x <_\phi y$ if $x \neq y$ and x precedes y in ϕ. We write $\phi \supseteq \mathcal{N}$ (ϕ covers $\mathcal{N}$) when $fv(\mathcal{N}) \subseteq dom(\phi)$ and $fn(\mathcal{N}) \subseteq na(\phi)$; similarly for $\phi \supseteq \mathcal{A}$ and $\phi \supseteq \mathcal{T}$.

Next we define three equivalence relations between name expressions, $\sim_N$, tree types, $\sim_T$, and high types, $\sim_H$ (often omitting the subscripts), and an apartness relation on name expressions, #. These relations are all indexed by a freshness signature that is understood to cover the free variables and names occurring in the expressions involved.

6-2 Definition: Equivalence and Apartness

$\mathcal{N} \sim_{N\phi} \mathcal{M}$ (a congruence, abbrev. $\mathcal{N} \sim_\phi \mathcal{M}$), and $\mathcal{N} \#_\phi \mathcal{M}$ (a symmetric relation) are the least such relations on name expressions such that $\phi \supseteq \mathcal{N},\mathcal{M}$, and:

$n \neq m \;!\; n \#_\phi m$	(Apart Names)
$\phi(x)=\mathsf{H} \;!\; n \#_\phi x$	(Apart Name Var)
$x <_\phi y \wedge \phi(y)=\mathsf{H} \;!\; x \#_\phi y$	(Apart Vars)
$\mathcal{N}\#_\phi\mathcal{M} \wedge \mathcal{P}\sim_\phi\mathcal{P}' \wedge \mathcal{Q}\sim_\phi\mathcal{Q}' \;!\; \mathcal{N}\{\mathcal{P}\leftrightarrow\mathcal{Q}\} \#_\phi \mathcal{M}\{\mathcal{P}'\leftrightarrow\mathcal{Q}'\}$	(Apart Congr)
$\mathcal{N}\sim_\phi\mathcal{N}' \wedge \mathcal{N}'\#_\phi\mathcal{M}' \wedge \mathcal{M}'\sim_\phi\mathcal{M} \;!\; \mathcal{N}\#_\phi\mathcal{M}$	(Apart Equiv)
$\mathcal{N}\{\mathcal{N}\leftrightarrow\mathcal{M}\} \sim_\phi \mathcal{M}$	(EqN ↔ App)
$\mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}\} \sim_\phi \mathcal{N}$	(EqN ↔ Id)
$\mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi \mathcal{N}\{\mathcal{M}'\leftrightarrow\mathcal{M}\}$	(EqN ↔ Symm)
$\mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi \mathcal{N}$	(EqN ↔ Inv)
$\mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}\{\mathcal{P}\leftrightarrow\mathcal{P}'\} \sim_\phi \mathcal{N}\{\mathcal{P}\leftrightarrow\mathcal{P}'\}\{\mathcal{M}\{\mathcal{P}\leftrightarrow\mathcal{P}'\}\leftrightarrow\mathcal{M}'\{\mathcal{P}\leftrightarrow\mathcal{P}'\}\}$	(EqN ↔ ↔)
$\mathcal{N}\#_\phi\mathcal{M} \wedge \mathcal{N}\#_\phi\mathcal{M}' \;!\; \mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi \mathcal{N}$	(EqN ↔ Apart)

$\mathcal{A} \sim_{T\phi} \mathcal{B}$ (abbrev. $\mathcal{A} \sim_\phi \mathcal{B}$), are the least relations on tree types such that $\phi \supseteq \mathcal{A},\mathcal{B}$ and:

they are congruences including α-conversion; we highlight:

$\mathcal{N} \sim_{N\phi} \mathcal{N}'$ and $\mathcal{A} \sim_\phi \mathcal{A}'$! $\mathcal{N}'[\mathcal{A}'] \sim_\phi \mathcal{N}'[\mathcal{A}']$ (EqT $\mathcal{N}[]$ Congr)

$\mathcal{A} \sim_{(\phi,\mathsf{H}x)} \mathcal{A}'$! $\mathsf{H}x.\,\mathcal{A} \sim_\phi \mathsf{H}x.\,\mathcal{A}'$ (EqT H Congr)

$\mathsf{H}x.\,\mathcal{A} \sim_\phi \mathsf{H}y.\,\mathcal{A}\{x \leftarrow y\}$ with $y \notin fv(\mathcal{A})$ (EqT H-α)

they distribute transpositions over all type constructors; we highlight:

$\mathbf{0}(\mathcal{M} \leftrightarrow \mathcal{M}') \sim_\phi \mathbf{0}$ (EqT $\mathbf{0}$ $\leftrightarrow$)

$(\mathcal{N}[\mathcal{A}])(\mathcal{M} \leftrightarrow \mathcal{M}') \sim_\phi \mathcal{N}(\mathcal{M} \leftrightarrow \mathcal{M}')[\mathcal{A}(\mathcal{M} \leftrightarrow \mathcal{M}')]$ (EqT $\mathcal{N}[]$ $\leftrightarrow$)

$(\mathsf{H}x.\mathcal{A})(\mathcal{M} \leftrightarrow \mathcal{M}') \sim_\phi$ (EqT H $\leftrightarrow$)

$\mathsf{H}x.(\mathcal{A}\{x \leftarrow x(\mathcal{M} \leftrightarrow \mathcal{M}')\}(\mathcal{M} \leftrightarrow \mathcal{M}'))$ with $x \notin fv(\mathcal{M},\mathcal{M}')$

$\mathcal{F} \sim_{H\phi} \mathcal{G}$, (abbrev. $\mathcal{F} \sim_\phi \mathcal{G}$), are the least relations on high types such that $\phi \supseteq \mathcal{F},\mathcal{G}$, and:

they are congruences including α-conversion; we highlight the cases for Π:

$\mathcal{F} \sim_{(\phi,\forall x)} \mathcal{G}$! $\Pi x.\,\mathcal{F} \sim_\phi \Pi x.\,\mathcal{G}$ (EqH Π Congr)

$\Pi x.\,\mathcal{G} \sim_{H\phi} \Pi y.\,\mathcal{G}\{x \leftarrow y\}$ with $y \notin fv(\mathcal{G})$ (EqH Π-α)

A notion of apartness of names from types is not necessary, since transpositions on types can be distributed down to transpositions on name expressions.

6-3 Definition: Valuation.

If ε:*Var*$\rightarrow\Lambda$ is a finite map, then we say that ε is a *valuation*.

We indicate by $\varepsilon(\mathcal{N})$, $\varepsilon(\mathcal{A})$ the homomorphic extensions of ε to name expressions and tree types, with the understanding that in such extension $\varepsilon(x) = x$ for $x \notin dom(\varepsilon)$.

If $fv(\mathcal{N}) \subseteq dom(\varepsilon)$ then we say that ε is a *ground valuation* for $\mathcal{N}$, and we write ε *grounds* $\mathcal{N}$; similarly for $\mathcal{A}$ and $\mathcal{F}$.

We say that a valuation ε satisfies a freshness signature ϕ if it respects the freshness constraints of ϕ, in the following sense:

6-4 Definition: Freshness Signature Satisfaction

$\varepsilon \vDash \phi$ iff $dom(\varepsilon) \subseteq dom(\phi)$

and $\forall x \in dom(\varepsilon).\ \phi(x) = \mathsf{H}$! $\varepsilon(x) \notin na(\phi)$

and $\forall y \in dom(\varepsilon).\ (x <_\phi y \wedge \phi(y) = \mathsf{H})$! $(x \in dom(\varepsilon) \wedge \varepsilon(x) \neq \varepsilon(y))$

In $\varepsilon \vDash \phi$ we do not require $dom(\phi) \subseteq dom(\varepsilon)$, to allow for partial valuations. But we require any partial valuation that instantiates an H variable to instantiate all the variables to the left of it (with distinct names).

The following soundness result requires some careful build-up: lemmas for instantiations of equivalence and apartness under partial valuations, for closure of satisfaction under closed equivalence, and substitution lemmas. We omit the details.

6-5 Proposition: Soundness of Equivalence and Apartness.

If $\mathcal{N} \#_\phi \mathcal{M}$ then $\forall \varepsilon \vDash \phi.\ (\varepsilon$ *grounds* $\mathcal{N},\mathcal{M})$! $\varepsilon(\mathcal{N}) \neq \varepsilon(\mathcal{M})$.

If $\mathcal{N} \sim_\phi \mathcal{M}$ then $\forall \varepsilon \vDash \phi.\ (\varepsilon$ *grounds* $\mathcal{N},\mathcal{M})$! $\varepsilon(\mathcal{N}) = \varepsilon(\mathcal{M})$.

If $\mathcal{A} \sim_\phi \mathcal{B}$ then $\forall \varepsilon \vDash \phi.\ (\varepsilon$ *grounds* $\mathcal{A},\mathcal{B})$! $\forall P.\ P \vDash \varepsilon(\mathcal{A})$! $P \vDash \varepsilon(\mathcal{B})$.

If $\mathcal{F} \sim_\phi \mathcal{G}$ then $\forall \varepsilon \vDash \phi.\ (\varepsilon$ *grounds* $\mathcal{F},\mathcal{G})$! $\forall F.\ F \vDash \varepsilon(\mathcal{F})$! $F \vDash \varepsilon(\mathcal{G})$.

7 Type System

We now present a type system that is sound for the operational semantics of Section 5. Subtyping includes the transposition equivalence of Section 6 (see rule (Sub Equiv)), and an unspecified collection of *ValidEntailments* that may capture aspects of logical implication. Apart from the flexibility given by subtyping through rule (Subsumption), the type rules for

terms are remarkably straightforward and syntax-driven.

The type system uses environments E that have a slightly unusual structure. They are ordered lists of either names (covering all the names occurring in expressions; see rule (NExpr n)), or variables (covering all the free variables of expressions; see rule (Term x)). Variables have associated type and freshness information of the form $x{:}\mathcal{T}$ if $\mathcal{T}{\neq}\mathbf{N}$, and $\mathsf{Q}x{:}\mathcal{T}$ (either $\forall x{:}\mathcal{T}$ or $\mathsf{H}x{:}\mathcal{T}$) if $\mathcal{T}{=}\mathbf{N}$. We write $dom(E)$ and $na(E)$ for the set of variables and the set of names defined by E. We write $E,x{:}\mathcal{T}$ and $E,\mathsf{Q}x{:}\mathbf{N}$ for the extension of E with a new association (provided that $x \notin dom(E)$), where $\mathcal{T}$ may depend on $dom(E)$. We write $E(x)$ for the (open) type associated to $x \in dom(E)$ in E. Moreover, in Definition 7-1 below we extract the freshness signature associated with an environment:

7-1 Definition: Freshness Signature of an Environment

$$fs(\emptyset) \triangleq \emptyset \qquad fs(E, \mathsf{Q}x{:}\mathbf{N}) \triangleq fs(E), \mathsf{Q}x$$
$$fs(E, n) \triangleq fs(E), n \qquad fs(E, x{:}\mathcal{T}) \triangleq fs(E)$$

Through $fs(E)$, in typing rule (Sub Equiv), typing environments are connected to the freshness signatures used in transposition equivalence.

7-2 Definition: Type Rules

Environments. Rules for $E \vdash \diamond$ (that is, E is well-formed).

$$\text{(Env }\emptyset\text{)}\ \frac{}{\emptyset \vdash \diamond} \qquad \text{(Env } n\text{)}\ \frac{E \vdash \diamond}{E, n \vdash \diamond} \qquad \text{(Env } x{\in}\mathbf{N}\text{)}\ \frac{E \vdash \diamond \quad \mathsf{Q}{\in}\{\forall,\mathsf{H}\} \quad x{\notin}dom(E)}{E, \mathsf{Q}x{:}\mathbf{N} \vdash \diamond}$$

$$\text{(Env } x{\notin}\mathbf{N}\text{)}\ \frac{E \vdash \diamond \quad E \vdash \mathcal{T} \quad \mathcal{T}{\neq}\mathbf{N} \quad x{\notin}dom(E)}{E, x{:}\mathcal{T} \vdash \diamond}$$

Names. Rules for $E \vdash_{\mathrm{N}} \mathcal{N}$ (that is, $\mathcal{N}$ is a name expression in E).

$$\text{(NExpr } n\text{)}\ \frac{E \vdash \diamond \quad n{\in}na(E)}{E \vdash_{\mathrm{N}} n} \qquad \text{(NExpr } x\text{)}\ \frac{E \vdash \diamond \quad E(x){=}\mathbf{N}}{E \vdash_{\mathrm{N}} x}$$

$$\text{(NExpr }{\leftrightarrow}\text{)}\ \frac{E \vdash_{\mathrm{N}} \mathcal{N} \quad E \vdash_{\mathrm{N}} \mathcal{M} \quad E \vdash_{\mathrm{N}} \mathcal{M}'}{E \vdash_{\mathrm{N}} \mathcal{N}(\mathcal{M}{\leftrightarrow}\mathcal{M}')}$$

Types. Rules for $E \vdash_{\mathrm{T}} \mathcal{A}$ and $E \vdash \mathcal{T}$ (that is, $\mathcal{A}$ is a tree type and $\mathcal{T}$ is a type in E). The rules are naturally syntax-driven, we highlight:

$$\text{(Type H)}\ \frac{E, \mathsf{H}x{:}\mathbf{N} \vdash_{\mathrm{T}} \mathcal{A}}{E \vdash_{\mathrm{T}} \mathsf{H}x.\mathcal{A}} \qquad \text{(Type }{\rightarrow}\text{)}\ \frac{E \vdash \mathcal{T} \quad E \vdash \mathcal{G} \quad \mathcal{T}{\neq}\mathbf{N}}{E \vdash \mathcal{T}{\rightarrow}\mathcal{G}} \qquad \text{(Type }\Pi\text{)}\ \frac{E, \forall x{:}\mathbf{N} \vdash \mathcal{G}}{E \vdash \Pi x.\, \mathcal{G}}$$

Subtyping. Rules for $E \vdash \mathcal{T} <: \mathcal{G}$ (that is, $\mathcal{T}$ is a subtype of $\mathcal{G}$ in E).

$$\text{(Sub Tree)}\ \frac{E \vdash_{\mathrm{T}} \mathcal{A} \quad E \vdash_{\mathrm{T}} \mathcal{B} \quad \langle \mathcal{A}, fs(E), \mathcal{B}\rangle \in ValidEntailments}{E \vdash \mathcal{A} <: \mathcal{B}} \qquad \text{(Sub Equiv)}\ \frac{E \vdash \mathcal{T} \quad E \vdash \mathcal{G} \quad \mathcal{T} \sim_{fs(E)} \mathcal{G}}{E \vdash \mathcal{T} <: \mathcal{G}}$$

$$\text{(Sub }\mathbf{N}\text{)}\ \frac{E \vdash \diamond}{E \vdash \mathbf{N} <: \mathbf{N}} \qquad \text{(Sub }{\rightarrow}\text{)}\ \frac{E \vdash \mathcal{T}' <: \mathcal{T} \quad E \vdash \mathcal{G} <: \mathcal{G}' \quad \mathcal{T},\mathcal{T}' {\neq} \mathbf{N}}{E \vdash \mathcal{T}{\rightarrow}\mathcal{G} <: \mathcal{T}'{\rightarrow}\mathcal{G}'} \qquad \text{(Sub }\Pi\text{)}\ \frac{E, \forall x{:}\mathbf{N} \vdash \mathcal{G} <: \mathcal{G}'}{E \vdash \Pi x.\, \mathcal{G} <: \Pi x.\, \mathcal{G}'}$$

Terms. Rules for $E \vdash t : \mathcal{T}$ (t has type $\mathcal{T}$ in E, with $E \vdash_{\mathrm{T}} t : \mathcal{A} \triangleq E \vdash_{\mathrm{T}} \mathcal{A} \wedge E \vdash t : \mathcal{A}$).

$$\text{(Term 0)}\ \frac{E \vdash \diamond}{E \vdash 0 : \mathbf{0}} \qquad \text{(Term }\mathcal{N}[]\text{)}\ \frac{E \vdash_{\mathrm{N}} \mathcal{N} \quad E \vdash_{\mathrm{T}} t : \mathcal{A}}{E \vdash \mathcal{N}[t] : \mathcal{N}[\mathcal{A}]} \qquad \text{(Term |)}\ \frac{E \vdash_{\mathrm{T}} t : \mathcal{A} \quad E \vdash_{\mathrm{T}} u : \mathcal{B}}{E \vdash t \mid u : \mathcal{A} \mid \mathcal{B}}$$

$$\text{(Term } \nu\text{)}\quad \frac{E,\ \mathsf{H}x{:}\mathbf{N} \vdash_{\mathrm{T}} t : \mathcal{A}}{E \vdash (\nu x)t : \mathsf{H}x.\mathcal{A}} \qquad \text{(Term } \leftrightarrow\text{)}\quad \frac{E \vdash_{\mathrm{T}} t : \mathcal{A} \quad E \vdash_{\mathrm{N}} \mathcal{M} \quad E \vdash_{\mathrm{N}} \mathcal{M}'}{E \vdash t(\mathcal{M}{\leftrightarrow}\mathcal{M}') : \mathcal{A}(\mathcal{M}{\leftrightarrow}\mathcal{M}')}$$

$$\text{(Term } \div\mathcal{N}[]\text{)}\quad \frac{E \vdash_{\mathrm{T}} t : \mathcal{N}[\mathcal{A}] \quad E,\ y{:}\mathcal{A} \vdash u : \mathcal{F}}{E \vdash t \div (\mathcal{N}[y{:}\mathcal{A}]).u : \mathcal{F}} \qquad \text{(Term } \div\mid\text{)}\quad \frac{E \vdash_{\mathrm{T}} t : \mathcal{A} \mid \mathcal{B} \quad E,\ x{:}\mathcal{A},\ y{:}\mathcal{B} \vdash u : \mathcal{F}}{E \vdash t \div (x{:}\mathcal{A} \mid y{:}\mathcal{B}).u : \mathcal{F}}$$

$$\text{(Term } \div\nu\text{)}\quad \frac{E \vdash_{\mathrm{T}} t : \mathsf{H}x.\mathcal{A} \quad E,\ \mathsf{H}x{:}\mathbf{N},\ y{:}\mathcal{A} \vdash_{\mathrm{T}} u : \mathcal{B}}{E \vdash t \div ((\nu x)y{:}\mathcal{A}).u : \mathsf{H}x.\mathcal{B}} \qquad \text{(Term ?)}\quad \frac{E \vdash_{\mathrm{T}} t : \mathcal{B} \quad E,\ x{:}\mathcal{A} \vdash u : \mathcal{F} \quad E,\ x{:}\neg\mathcal{A} \vdash v : \mathcal{F}}{E \vdash t?(x{:}\mathcal{A}).u,v : \mathcal{F}}$$

$$\text{(Term } x\text{)}\quad \frac{E \vdash \diamond \quad x{\in}dom(E)}{E \vdash x : E(x)} \qquad \text{(Term } \mathcal{N}\text{)}\quad \frac{E \vdash_{\mathrm{N}} \mathcal{N}}{E \vdash \mathcal{N} : \mathbf{N}} \qquad \text{(Term } \lambda\text{)}\quad \frac{E,\ x{:}\mathcal{F} \vdash t : \mathcal{G} \quad \mathcal{F}{\neq}\mathbf{N}}{E \vdash \lambda x{:}\mathcal{F}.t : \mathcal{F}{\rightarrow}\mathcal{G}} \qquad \text{(Term App)}\quad \frac{E \vdash t : \mathcal{F}{\rightarrow}\mathcal{G} \quad E \vdash u : \mathcal{F}}{E \vdash t(u) : \mathcal{G}}$$

$$\text{(Term Dep}\lambda\text{)}\quad \frac{E,\ \forall x{:}\mathbf{N} \vdash t : \mathcal{G}}{E \vdash \lambda x{:}\mathbf{N}.t : \Pi x.\ \mathcal{G}} \qquad \text{(Term DepApp)}\quad \frac{E \vdash t : \Pi x.\ \mathcal{G} \quad E \vdash_{\mathrm{N}} \mathcal{N}}{E \vdash t(\mathcal{N}) : \mathcal{G}\{x{\leftarrow}\mathcal{N}\}} \qquad \text{(Subsumption)}\quad \frac{E \vdash t : \mathcal{F} \quad E \vdash \mathcal{F} <: \mathcal{G}}{E \vdash t : \mathcal{G}}$$

Notes: • As we already mentioned, the type system includes dependent types, with binding operators $\mathsf{H}x.\mathcal{A}$ (the type of hiding in trees) and $\Pi x.\mathcal{F}$ (the type of those functions $\lambda x{:}\mathbf{N}.t$ such that the type $\mathcal{F}$ of t may depend on the input variable x).

• The subtyping relation is parameterized by a set *ValidEntailments*, assumed to consist of triples $\langle\mathcal{A},\phi,\mathcal{B}\rangle$ that are sound ($\forall\varepsilon \vDash\phi.\ (\varepsilon\ grounds\ \mathcal{A},\mathcal{B})\ !\quad \forall P.\ P \vDash \varepsilon(\mathcal{A})\ !\quad P \vDash \varepsilon(\mathcal{B})$).

• The use of $x{:}\neg\mathcal{A}$ in (Term ?) means $x{:}\mathcal{A}$! **F**, but this assumption is not very useful without a rich theory of subtyping: see discussion in Section 1.3. On the other hand, there are no significant problems in executing run-time type tests such as $P \vDash \neg\mathcal{A}$ (see Definition 4-1), e.g., resulting from $t?(x{:}\neg\mathcal{A}).u,v$. A more informative typing of x for the third assumption of this rule is $x{:}\mathcal{B}\wedge\neg\mathcal{A}$, but we lack a compelling use for it.

• In (Term $\div\mathcal{N}$[]) (and (Term $\div\mid$), (Term ?)) we do not need extra assumptions $E \vdash \mathcal{F}$ to avoid the escape of y (and x,y, and x) into $\mathcal{F}$, because these are not variables of type **N**, and $\mathcal{F}$ cannot depend on them. We do not need the extra assumption in (Term $\div\nu$) for x because there we rebind the result type.

• In (TermDepApp) we require the argument $\mathcal{N}$ to be a name expression, not an expression of type **N**, so we can do a substitution $\mathcal{G}\{x{\leftarrow}\mathcal{N}\}$ into the type. Note that $E \vdash t : \mathbf{N}$ means that t can be any computation of type **N**, unlike $E \vdash_{\mathrm{N}} \mathcal{N}$.

A stack satisfies an environment, $\rho \vDash E$, if $\rho(x) \vDash \rho(E(x))$ for all x's in $dom(E)$; note the extra $\rho(\text{-})$ used to bind the dependent variables in $E(x)$. Here $\rho(\mathcal{F})$ or $\rho(\mathcal{A})$ means that ρ is used as a valuation (Definition 6-3). Moreover, we require ρ to satisfy the freshness signature extracted from E. We write $\rho\backslash x$ for the restriction of ρ to $dom(\rho)$-$\{x\}$.

7-3 Definition: Environment Satisfaction

$\rho \vDash E$ iff $dom(E){\subseteq}dom(\rho)$
and $\rho \vDash fs(E)$ (where ρ is seen as a valuation ε; see Definition 6-4)
and $\forall x{\in}dom(E).\ \rho(x) \vDash \rho(E(x))$

7-4 Theorem: Subject Reduction.
(1) If $E \vdash \mathcal{T} <: \mathcal{G}$ and $\rho \vDash E$ and $F \vDash_H \rho(\mathcal{T})$ then $F \vDash_H \rho(\mathcal{G})$.
(2) If $E \vdash_N \mathcal{N}$ and $\rho \vDash E$ and $\mathcal{N} \downarrow_\rho n$ then $n \vDash_N \rho(\mathcal{N})$.
(3) If $E \vdash t : \mathcal{T}$ and $\rho \vDash E$ and $t \forall_\rho F$, then $F \vDash_H \rho(\mathcal{T})$.

Proof

We show the (Term ν) case of (3), which is by induction on the derivation of $E \vdash t : \mathcal{T}$. We have $E \vdash (\nu x)t : \mathsf{H}x.\mathcal{A}$ and $\rho \vDash E$ and $(\nu x)t \forall_\rho F$. We have from (Term ν) $E, \mathsf{H}x{:}\mathbf{N} \vdash_T t : \mathcal{A}$, and from (Red ν) $F = (\nu n)P$ and $t \forall_{\rho[x\leftarrow n]} P$ for $n \notin na(t,\rho)$. Since n could appear in $\mathcal{A}$, blocking the last step of this proof, take $n' \notin na(t,\mathcal{A},\rho,P)$, so that $F \equiv_\alpha (\nu n')P\bullet(n\leftrightarrow n')$. By Lemma 5-2 $t\bullet(n\leftrightarrow n') \forall_{\rho[x\leftarrow n]\bullet(n\leftrightarrow n')} P\bullet(n\leftrightarrow n')$, that is $t \forall_{\rho[x\leftarrow n']} P\bullet(n\leftrightarrow n')$. We have $\rho[x\leftarrow n'] \vDash E, \mathsf{H}x{:}\mathbf{N}$. By Ind Hyp, $P\bullet(n\leftrightarrow n') \vDash \rho[x\leftarrow n'](\mathcal{A})$, that is, $P\bullet(n\leftrightarrow n') \vDash \rho\backslash x(\mathcal{A})\{x\leftarrow n'\}$. Since $F \equiv (\nu n')P\bullet(n\leftrightarrow n')$ and $n' \notin na(\rho\backslash x(\mathcal{A}))$, by Definition 4-1, $F \vDash \mathsf{H}x.\rho\backslash x(\mathcal{A})$. That is, $F \vDash \rho(\mathsf{H}x.\mathcal{A})$. □

8 Examples

We discuss some programming examples, using plausible (but not formally checked) extensions of the formal development of the previous sections. In particular, we use recursive types, ***rec*** *X*. $\mathcal{A}$, existential types $\exists x.\mathcal{A}$ where x ranges over names (these are simpler to handle than $\mathsf{H}x.\mathcal{A}$), and a variant of location matching, $t \div (\underline{x}[\underline{y}{:}\mathcal{A}]).u$, that binds labels x from the data in addition to contents y (its typing requires existential types). Examples of transposition types have been discussed in the Introduction; here we concentrate on pattern matching, using some abbreviations:

test *t* ***as*** *w*:$\mathcal{A}$ ***then*** *u* ***else*** *v*	for	$t?(w{:}\mathcal{A})$. *u*, *v*
match *t* ***as*** (*pattern*) ***then*** *u* ***else*** *v*	for	$t?(w{:}\mathcal{B})$. $(w \div (pattern).u)$, *v*

where $\mathcal{B}$ is the type naturally extracted from *pattern*.

We also use nested patterns, in the examples, which can be defined in a similar way. We use standard notations for recursive function definitions. We sometimes underline binding occurrences of variables, for clarity. We explicitly list the subtypings, if any, that must be included in *ValidEntailments* for these examples to typecheck (none are needed for the examples in Section 1.2).

Basic. Duplicating a given label, and duplicating a hidden label:

$\lambda x{:}\mathbf{N}.\ \lambda y{:}x[\mathbf{T}].\ x[y]$: $\Pi x.\ (x[\mathbf{T}] \to x[x[\mathbf{T}]])$
$\lambda z{:}(\mathsf{H}x.x[\mathbf{T}]).\ z \div ((\nu x)y{:}x[\mathbf{T}]).\ x[y]$: $(\mathsf{H}x.x[\mathbf{T}]) \to (\mathsf{H}x.x[x[\mathbf{T}]])$

Collect. Collect all the subtrees that satisfy $\mathcal{A}$, even under restrictions:

let type *Result* = ***rec*** *X*. $\mathbf{0} \vee \mathcal{A} \vee (X \mid X) \vee \mathsf{H}x.X$

let rec *collect*($\underline{x}$: **T**): *Result* =
(***test*** *x* ***as*** $\underline{w}{:}\mathcal{A}$ ***then*** *w* ***else*** 0) |
(***test*** *x* ***as*** $\underline{w}{:}\mathbf{0}$ ***then*** 0 ***else***
match *x* ***as*** ($\underline{y}{:}\neg\mathbf{0} \mid \underline{w}{:}\neg\mathbf{0}$) ***then*** *collect*(*y*) | *collect*(*w*) ***else***
match *x* ***as*** ($\underline{y}[\underline{w}{:}\mathbf{T}]$) ***then*** *collect*($\underline{w}$) ***else***
match *x* ***as*** $((\nu\underline{y})\underline{w}{:}\copyright y)$ ***then*** *collect*(*w*) ***else*** 0)

Recall that, in the last match, a (νy) is automatically wrapped around the result; hence the $\mathsf{H}x.X$ in the definition of *Result*. The typing $\underline{w}{:}\copyright y$ (instead of $\underline{w}{:}\mathbf{T}$) is used to reduce nondeterminism by forcing the analysis of non-redundant restrictions. Similarly, the pattern $\neg\mathbf{0} \mid \neg\mathbf{0}$ is used to avoid vacuous splits where one component is 0. In general, the splitting of composition is nondeterministic; in this case the result may or may not be uniquely determined de-

pending on the shape of $\mathcal{A}$. The subtypings needed here are $\mathcal{A} <: \mathbf{T}$, $\mathcal{A} <: \mathcal{A} \vee \mathcal{B}$ and ***rec*** fold/unfold.

Removing Dangling Pointers. We can encode addresses and pointers in the same style as in XML. An address definition is encoded as *addr*[*n*[0]], where *addr* is a conventional name, and *n* is the name of a particular address. A pointer to an address is encoded as *ptr*[*n*[0]], where *ptr* is another conventional name. Addresses may be global, like URLs, or local, like XML's IDs; local addresses are represented by restriction: (ν*n*) ... *addr*[*n*[0]] ... *ptr*[*n*[0]] ... . A tree should not contain two address definitions for the same name, but this assumption is not important in our example.

We write a function that copies a tree, including both public and private addresses, but deletes all the pointers that do not have a corresponding address in the tree. Every time a *ptr*[*n*[0]] is found, we need to see if there is an *addr*[*n*[0]] somewhere in the tree. But we cannot search for *addr*[*n*[0]] in the original tree, because *n* may be a restricted address we have come across. So, we first open all the (non-trivial) restrictions, and then we proceed as above, passing the root of the restriction-free tree as an additional parameter. The search for *addr*[*n*[0]] can be done by a single type test for *Somewhere*(*addr*[*n*[**0**]]), where *Somewhere*($\mathcal{A}$) ≜ ***rec*** *X*. ($\mathcal{A}$ | **T**) ∨ ∃*y*.(*y*[*X*] | **T**).

let rec *deDangle*(*x*: **T**): **T** =
  match *x* ***as*** ((ν*y*)*w*:©*y*) ***then*** *deDangle*(*w*) ***else*** *f* (*x*, *x*)
and *f* (*x*: **T**, *root*: **T**): **T** =
  test *x* ***as*** *w*:**0** ***then*** 0 ***else***
  match *x* ***as*** (*y*:¬**0** | *w*:¬**0**) ***then*** *f* (*y*, *root*) | *f* (*w*, *root*) ***else***
  match *x* ***as*** (*ptr*[*y*[**0**]]) ***then***
    test *root* ***as*** *w*:*Somewhere*(*addr*[*y*[**0**]]) ***then*** *ptr*[*y*[0]] ***else*** 0 ***else***
  match *x* ***as*** (*z*[*w*:**T**]) ***then*** *z*[*f* (*w*, *root*)] ***else*** 0

Note that *deDangle* automatically recloses, in the result, all the restrictions that it opens. The subtypings needed here are just $\mathcal{A} <: \mathbf{T}$.

9 Conclusions and Acknowledgments

We have introduced a language and a rich type system for manipulating semistructured data with hidden labels and scope extrusion, via pattern matching and transpositions.

As advocated in [23,30], our formal development could be carried out within a metatheory with transpositions; then, Lemmas 4-3 and 5-2 would fall out of the metatheory, and one could be less exposed to mistakes in α-conversion issues. We have not gone that far, but we should seriously consider this option in the future.

We are not aware of previous uses of transpositions in structural operational semantics, although this falls within the general framework of [30]. We believe ours is the first calculus or language with explicit transpositions in the syntax of terms and types.

Thanks to Murdoch J. Gabbay for illuminating discussions on transpositions, and to Luís Caires who indirectly influenced this paper through earlier work with the first author. Moreover, Gabbay and Caires helped simplify the technical presentation.

References

[1] S. Abiteboul, P. Buneman, D. Suciu.: **Data on the Web**. Morgan Kaufmann Publishers, 2000.
[2] S. Abiteboul, P. Kanellakis: **Object identity as a query language primitive.** Journal of the ACM, 45(5):798-842, 1998. A first version appeared in SIGMOD'89.
[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. **The Lorel Query Language for**

Semistructured Data. International Journal on Digital Libraries, 1(1), pp. 68-88, April 1997.

[4] M.P. Atkinson, F. Bancilhon, et al.: **The Object-Oriented Database System Manifesto.** Building an Object-Oriented Database System, The Story of O2, 1992, pp. 3-20.

[5] V. Benzaken, G. Castagna, A. Frisch: **CDuce: a white paper.** PLAN-X: Programming Language Technologies for XML, Pittsburgh PA, Oct. 2002. http://www.cduce.org.

[6] S. Boag, D. Chamberlin, M.F. Fernandez, D. Florescu, J. Robie, J. Siméon: **XQuery 1.0: An XML Query Language,** W3C Working Draft, 2002, http://www.w3.org/TR/xquery.

[7] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler: **Extensible Markup Language (XML) 1.0 (Second Edition)**, W3C document, http://www.w3.org/TR/REC-xml.

[8] P. Buneman, S.B. Davidson, G.G. Hillebrand, D. Suciu: **A Query Language and Optimization Techniques for Unstructured Data.** SIGMOD Conference 1996, pp. 505-516.

[9] L. Caires: **A Model for Declarative Programming and Specification with Concurrency and Mobility**. Ph.D. Thesis, Dept. de Informática, FTC, Universidade Nove de Lisboa, 1999.

[10] L. Caires, L Cardelli: **A Spatial Logic for Concurrency: Part I**. Proc. TACS 2001, Naoki Kobayashi and Benjamin C. Pierce (Eds.). LNCS. 2215. Springer, 2001, pp 1-37. To appear in I&C.

[11] L. Caires, L. Cardelli: **A Spatial Logic for Concurrency: Part II**. Proc. CONCUR'02, 2002.

[12] C. Calcagno, L. Cardelli, A.D. Gordon, **Deciding Validity in a Spatial Logic for Trees**. Draft.

[13] C. Calcagno, H. Yang, P.W. O'Hearn: **Computability and Complexity Results for a Spatial Assertion Language for Data Structures**. Proc. FSTTCS 2001, pp. 108-119.

[14] L. Cardelli, P. Gardner, G. Ghelli, **A Spatial Logic for Querying Graphs**. Proc. ICALP'02, Peter Widmayer et al. (Eds.). LNCS 2380, Springer, 2002. pp 597-610.

[15] L. Cardelli, G. Ghelli, **A Query Language Based on the Ambient Logic**. Proc. ESOP'01, David Sands (Ed.). LNCS 2028, Springer, 2001, pp. 1-22.

[16] L. Cardelli, A.D. Gordon, **Anytime, Anywhere. Modal Logics for Mobile Ambients.** Proc. of the 27th ACM Symposium on Principles of Programming Languages, 2000, pp. 365-377.

[17] S. Cluet, S. Jacqmin, and J. Simeon. **The New YATL: Design and Specifications.** INRIA, 1999.

[18] E. Cohen: **Validity and Model Checking for Logics of Finite Multisets.** Draft.

[19] D. Florescu, A. Deutsch, A. Levy, D. Suciu, M. Fernandez: **A Query Language for XML**. In Proc. of Eighth International World Wide Web Conference, 1999.

[20] D. Florescu, A. Levy, M. Fernandez, D. Suciu, **A Query Language for a Web-Site Management System.** SIGMOD Record , vol. 26 , no. 3 , pp. 4-11 , September, 1997.

[21] M.J. Gabbay: **A Theory of Inductive Definitions with a-Equivalence: Semantics, Implementation, Programming Language.** Ph.D. Thesis, University of Cambridge, 2000.

[22] M.J. Gabbay, A.M. Pitts, **A New Approach to Abstract Syntax Involving Binders**. Proc. LICS1999. IEEE Computer Society Press, 1999. pp 214-224.

[23] M.J. Gabbay: **FM-HOL, A Higher-Order Theory of Names.** In Thirty Five years of Automath, Heriot-Watt University, Edinburgh, April 2002. Inforal Proc., 2002.

[24] A.D. Gordon: **Notes on Nominal Calculi for Security and Mobility.** R.Focardi, R.Gorrieri (Eds.): Foundations of Security Analysis and Design. LNCS 2171. Springer, 1998.

[25] A.D. Gordon, A. Jeffrey: **Typing Correspondence Assertions for Communication Protocols.** MFPS 17, Elsevier Electronic Notes in Theoretical Computer Science, Vol 45, 2001.

[26] H. Hosoya, B. C. Pierce: **XDuce: A Typed XML Processing Language (Preliminary Report)**. WebDB (Selected Papers) 2000, pp: 226-244

[27] R. Milner: **Communicating and Mobile Systems: the π-Calculus**. Cambridge U. Press, 1999.

[28] P.W. O'Hearn, D. Pym: **Logic of Bunched Implication.** Bulletin of Symbolic Logic 5(2), pp 215-244, 1999.

[29] P.W. O'Hearn, J.C. Reynolds, H. Yang: **Local Reasoning about Programs that Alter Data Structures**. Proc. CSL 2001, pp. 1-19.

[30] A.M. Pitts: **Nominal Logic, A First Order Theory of Names and Binding.** Proc. TACS 2001, Naoki Kobayashi and Benjamin C. Pierce (Eds.). LNCS 2215. Springer, 2001, pp 219-242.

[31] A.M. Pitts, M.J. Gabbay: **A Metalanguage for Programming with Bound Names Modulo Renaming.** R. Backhouse and J.N. Oliveira (Eds.): MPC 2000, LNCS 1837, Springer, pp. 230-255.

The Converse of a Stochastic Relation

Ernst-Erich Doberkat

Chair for Software Technology
University of Dortmund
doberkat@acm.org

Abstract. Transition probabilities are proposed as the stochastic counterparts to set-based relations. We propose the construction of the converse of a stochastic relation. It is shown that two of the most useful properties carry over: the converse is idempotent as well as anticommutative. The nondeterminism associated with a stochastic relation is defined and briefly investigated. We define a bisimulation relation, and indicate conditions under which this relation is transitive; moreover it is shown that bisimulation and converse are compatible.
Keywords: Stochastic relations, concurrency, bisimulation, converse, relational calculi, nondeterminism.

1 Introduction

The use of relations is ubiquitous in Mathematics, Logic and Computer Science, their systematic study goes back as far as Schröder's seminal work. Ongoing research with a focus on program specification may be witnessed from the wealth of material collected in [18, 4]. The map calculus [5] shows that these methods determine an active line of research in Logic.

This paper deals with stochastic rather than set-valued relations, it studies the converse of such a relation. It investigates furthermore some similarities between forming the converse for set-theoretic relations and for their stochastic cousins.

For introducing into the problem, let R be a relation, i.e., a set of pairs of, say, states. If $\langle x, y\rangle \in R$, then this is written as $x \rightarrow_R y$ and interpreted as a state transition from x to y. The converse $R^\smile$ shifts attention to the goal of the transition: $y \rightarrow_{R^\smile} x$ is interpreted as y being the goal of a transition from x. Now let $p(x, y)$ be the probability that there is a transition from x to y, and the question arises with which probability state y is the goal of a transition from x. This question cannot be answered unless we know the initial probabilities for the states. Then we can calculate $p_\mu^\smile(y, x)$ as the probability to make a transition from x to y weighted by the probability to start from x conditional to the event to reach y at all, i.e.

$$p_\mu^\smile(y, x) := \frac{\mu(x) \cdot p(x, y)}{\sum_t \mu(t) \cdot p(t, y)}.$$

Consider as an example the simple transition system p on three states given in the left hand side of Fig. 1. The converse $p_\mu^\smile$ for the initial probability $\mu :=$ [1/2 1/4 1/4] is given on the right hand side.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 233–249, 2003.

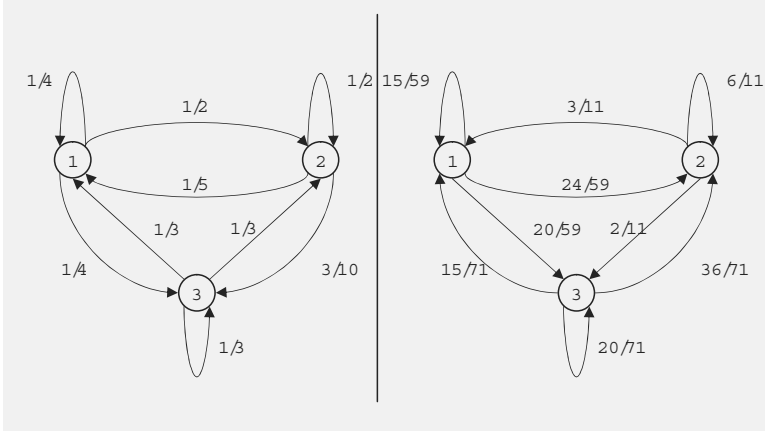


Fig. 1. A Stochastic Relation and Its Converse

The situation is more complicated in the non-finite case, which is considered here; since some measure theoretic constructions do not work in the general case, we assume that the measurable structure comes from Polish, i.e., second countable and completely metrizable topological spaces (like the real line $\mathbb{R}$). A definition of the converse $K_\mu^\smile$ of a stochastic relation K given an initial distribution μ is proposed in terms of disintegration. An interpretation of the converse in terms of random variables is given, and it is shown that the converse behaves with respect to composition like its set-theoretic counterpart, viz., $(K;L)_\mu^\smile = L_{K^\bullet(\mu)}^\smile ; K_\mu^\smile$, where $K^\bullet(\mu)$ denotes the image distribution of μ under K, and the composition is the Kleisli composition for the corresponding monad (section 4). This is of course the probabilistic counterpart to the corresponding law for relations R and S, which reads $(R;S)^\smile = S^\smile ; R^\smile$.

The set $\{K_\mu^\smile(y) | y \in Y\}$ of all sub-probability measures constituting the converse turns out to have an interesting property: it is topologically rather small, i.e., its closure is compact in the weak topology of sub-probability measures on Y (Cor. 4.2). This indicates that the converse $K_\mu^\smile$ does not carry as much information as K or μ do.

A stochastic relation K between X and Y induces a set-theoretic relation R_K (called the *fringe relation*) in the following way: let $\langle x, y\rangle \in R_K$ iff $K(x)(U) > 0$ for each open neighborhood U of y. Relation R_K is considered as K's nondeterminism, since it indicates the set of all pairs that are possible for the stochastic relation K. The relationship between these relations is briefly investigated in terms of natural transformations between two functors in Sect. 3.

A stochastic relation models the dynamics of a system, which is partly captured through the notion of bisimilarity. Thus the question of stability under bisimilarity arises when constructing the converse. We define in section 5 a suitable notion of bisimilarity and show that this is a transitive relation. It is shown

that the converses $K_{\mu}^{\smile}$ and $L_{\nu}^{\smile}$ are bisimilar, provided K and L as well as the initial distributions μ and ν are bisimilar[1].

Acknowledgements. Part of this work could be done while the author was visiting the Dipartimento di Informatica at the University of L'Aquila. The visit was in part supported through grants from the Exchange Programme for Scientists between Italy and Germany from the Italian Minstry of Foreign Affairs/Deutscher Akademischer Austauschdienst and from Progetto speciale I.N.D.A.M./GNIM *Nuovi paradigmi di calcolo: Linguaggi e Modelli*. The author wants to thank Eugenio Omodeo and Gunther Schmidt for getting him interested in relational methods. The referees' comments and suggestions are gratefully acknowledged.

2 Stochastic Relations

Before stochastic relations are introduced, some basic facts from measure theory are recalled. We also introduce some basic operations on these relations.

A *Polish space* $(X, \mathcal{T})$ is a topological space which has a countable dense subset, and which is metrizable through a complete metric. The *Borel sets* $\mathcal{B}(X, \mathcal{T})$ for the topology $\mathcal{T}$ is the smallest σ-algebra on X which contains $\mathcal{T}$. A *Standard Borel* space $(X, \mathcal{A})$ is a measurable space such that the σ-algebra $\mathcal{A}$ equals $\mathcal{B}(X, \mathcal{T})$ for some Polish topology $\mathcal{T}$ on X. Although the Borel sets are determined uniquely through the topology, the converse does not hold, as we will see in a short while. Given two measurable spaces $(X, \mathcal{A})$ and $(Y, \mathcal{B})$, a map $f : X \to Y$ is $\mathcal{A} - \mathcal{B}$-*measurable* whenever $f^{-1}[\mathcal{B}] \subseteq \mathcal{A}$ holds, where $f^{-1}[\mathcal{B}] := \{f^{-1}[B] | B \in \mathcal{B}\}$ is the set of inverse images $f^{-1}[B] := \{x \in X | f(x) \in B\}$ of elements of $\mathcal{B}$. If the σ-algebras are the Borel sets of some topologies on X and Y, resp., then a measurable map is called *Borel measurable* or simply a *Borel* map. The real numbers $\mathbb{R}$ carry always the Borel structure induced by the usual topology which will not be mentioned explicitly when talking about Borel maps.

The category $\mathfrak{SB}$ has as objects Standard Borel (SB) spaces, a morphism $f \in \mathfrak{SB}(X, Y)$ between two SB spaces X and Y is a Borel map $f : X \to Y$.

Recall that a map $f : X \to Y$ between the topological spaces $(X, \mathcal{T})$ and $(Y, \mathcal{S})$ is *continuous* iff the inverse image of an open set from $\mathcal{S}$ is an open set in $\mathcal{T}$. Thus a continuous map is also measurable with respect to the Borel sets generated by the respective topologies.

When the context is clear, we will write down Polish spaces without their topologies, and the Borel sets are always understood with respect to the topology. $\mathcal{M}(X)$ denotes the vector space of all bounded real-valued Borel maps on the SB-space X.

The set $\mathbf{S}(X)$ denotes the set of all sub-probability measures on the SB space X. The former set carries the weak topology, i.e., the smallest topology which makes the map $\mu \mapsto \int_X f \, d\mu$ for all continuous functions $f : X \to \mathbb{R}$ continuous as soon as X carries a Polish topology. It is well known that the

[1] The full paper is available as `Technische-Berichte/Doberkat_SWT-Memo-113-ps.gz` in directory `ftp://ls10-www.cs.uni-dortmund.de/pub`

weak topology on $\mathbf{S}(X)$ is a Polish space [16, Theorem II.6.5], and that its Borel sets are the smallest σ-algebra on $\mathbf{S}(X)$ for which for any Borel set $B \subseteq Y$ the map $\mu \mapsto \mu(B)$ is measurable. This σ-algebra is sometimes called the *weak-*-σ-algebra* in stochastic dynamic optimization. Note that the weak-*-σ-algebra depends only on the σ-algebra of the underlying SB-space, hence is independent of any specific Polish topology. An argument due to Giry [11] shows that $\mathbf{S}$ is the functorial part of a monad over $\mathfrak{SB}$, and that the Kleisli morphisms coming with this monad are just the stochastic relations.

Given two Polish spaces X and Y, a *stochastic relation* $K : X \rightsquigarrow Y$ is a Borel map from X to the set $\mathbf{S}(Y)$. Hence $K : X \rightsquigarrow Y$ is a stochastic relation iff

1. $K(x)$ is a sub-probability measure on (the Borel sets of) Y for all $x \in X$,
2. $x \mapsto K(x)(B)$ is a measurable map for each Borel set $B \subseteq Y$.

Composition of stochastic relations is the Kleisli composition: let $K : X \rightsquigarrow Y$ and $L : Y \rightsquigarrow Z$, then define for $x \in X, C \in \mathcal{B}_Z$:

$$(K;L)(x)(C) := \int_Y L(y)(C)\ K(x)(dy).$$

Standard arguments show that $K;L : X \rightsquigarrow Z$.

In terms of input/output systems, $K(x)(dy)$ may be interpreted that dy is the output of the system modelled by K after input x; the system does not need to be strictly probabilistic in the sense that each input produces an output with probability 1, i.e., $K(x)(Y) = 1$ does not hold necessarily. $K(x)(Y) < 1$ may occur when K models a non-terminating computation, so that $1 - K(x)(Y)$ is the probability for the event *no output at all*. Note that the Markov processes investigated in [6, 10] are special cases.

Example 1. In the discrete case a stochastic relation p between $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$ is represented through a non-negative substochastic matrix

$$(p(i,j))_{1 \le i \le n, 1 \le j \le m}.$$

The composition of two stochastic relations p and q is expressed through matrix multiplication, which is the discrete analogue to the Kleisli product above. $\diamond$

We collect some constructions and indicate some well known properties which will be helpful in the sequel. It shows how a measurable map and a measure induce a measure on the range of that map, and how a measure and a stochastic relation define a measure on the relation's target space, and on the product space, resp.

Definition 1. *Let X and Y be SB-spaces.*

1. *$f^\flat(\mu)(B) := \mu(f^{-1}[B])$ defines a map $\mathfrak{SB}(X,Y) \times \mathbf{S}(X) \to \mathbf{S}(Y)$ such that*

$$\int_Y g\ df^\flat(\mu) = \int_X g \circ f\ d\mu$$

holds for each $g \in \mathcal{M}(Y)$.

2. $K^\bullet(\mu)(B) := \int_X K(x)(B)\ \mu(dx)$ *defines a map* $\mathfrak{SB}(X, \mathbf{S}(Y)) \times \mathbf{S}(X) \to \mathbf{S}(Y)$ *such that*

$$\int_Y g\ dK^\bullet(\mu) = \int_X \int_Y g(y)\ K(x)(dy)\ \mu(dx)$$

holds for each $g \in \mathcal{M}(Y)$.
3. $(\mu \otimes K)(D) := \int_X K(x)(D_x)\ \mu(dx)$ *defines a map* $\mathbf{S}(X) \times \mathfrak{SB}(X, \mathbf{S}(Y)) \to \mathbf{S}(X \times Y)$ *such that*

$$\int_{X\times Y} g\ d(\mu \otimes K) = \int_Y \int_X g(x, y)\ K(x)(dy)\ \mu(dx)$$

is true whenever $g \in \mathcal{M}(X \times Y)$.

Since the integral in property 1 changes variables, it is sometimes referred to as the *Change of Variables formula*. Property 3 uses $D_x := \{y \in Y | \langle x, y\rangle \in D\}$ for the measurable set $D \subseteq X \times Y$; it gives the integral over a product as repeated integrals and contains the Fubini Theorem as special case.

Note that $f^\flat(\mu)$ is $\mathbf{S}(f)(\mu)$, the former notation being somewhat more light-handed; $K^\bullet$ is just the forgetful functor from the Kleisli category of the Giry monad, and the tensor construction in the third part arises from the tensorial strength of the monad.

Example 2. Illustrating these constructions through the discrete case, assume that $p : \{1, \ldots, n\} \rightsquigarrow \{1, \ldots, m\}$ is a stochastic relation, and let $\mu \in \mathbf{S}(\{1, \ldots, n\})$ be an initial distribution. Then

1. $f^\flat(\mu)(j) = \sum_{f(i)=j} \mu(i)$ is the probability that $f : \{1, \ldots, n\} \to \{1, \ldots, m\}$ hits the value j.
2. $p^\bullet(\mu)(j) = \sum_{i=1}^n \mu(i) \cdot p(i, j)$ is the probability that response j is produced, given the initial probability μ.
3. $(\mu \otimes p)(\langle i, j\rangle) = \mu(i) \cdot p(i, j)$ gives the probability for the input/output pair $\langle i, j\rangle$ to occur, given the initial probability μ (which is responsible for input i), and the probability $p(i, j)$ for output j after input i.

These properties are easily established using elementary computations. $\diamond$

Some properties of the general constructions are collected for the reader's convenience:

1. $(K;L);M = K;(L;M)$,
2. $(K;L)^\bullet = K^\bullet \circ L^\bullet$ (where $\circ$ denotes the usual composition of maps),
3. for $f \in \mathcal{M}(Z)$ and for $x \in X$ the equality

$$\int_Z f\ d(K;L)(x) = \int_Y \int_Z f(z)\ L(y)(dz)\ K(x)(dy)$$

holds.

4. $K;\mathbb{I}_Y = K$ and $\mathbb{I}_X;K = K$, where $\mathbb{I}_X : X \rightsquigarrow X$ is the unit kernel on X which is defined by

$$\mathbb{I}_X(x)(A) := \delta_x(A) := \texttt{if } x \in A \texttt{ then } 1 \texttt{ else } 0 \texttt{ fi}.$$

It is remarkable that the construction in Def. 1. 3 can be reversed, and this is in fact the cornerstone for constructing the converse of a stochastic relation, as will be seen in Sect. 4. Reversing the construction means that each measure on the product of two SB-spaces can be represented as a product from the measure of a measure and a stochastic relation (or, putting it in terms of Def.1: the map $\langle \mu, K\rangle \mapsto \mu \otimes K$ defined in part 3 between $\mathbf{S}(X) \times \mathfrak{SB}(X, \mathbf{S}(Y))$ and $\mathbf{S}(X \times Y)$ is onto).

Proposition 1. *Let X and Y be SB-spaces, and $\zeta \in \mathbf{S}(X \times Y)$. Then there exists a stochastic relation $K : X \rightsquigarrow Y$ such that $\zeta = \pi^{\flat}_{X\times Y,X}(\zeta) \otimes K$, $\pi_{X\times Y,X}$ denoting the projection from $X \times Y$ to X.*

Proof. [16, Theorem V.8.1]. □

The stochastic relation K is uniquely determined up to sets of μ-measure zero; it is known as the *regular conditional distribution of π_Y given π_X*, cf. [16, Ch. V.8]. We will call K *a version of the disintegration of ζ w.r.t. $\pi^{\flat}_{X\times Y,X}(\zeta)$*.

Example 3. Let $\zeta \in \mathbf{S}(\{1,\ldots,n\} \times \{1,\ldots,m\})$, then the probability $p(i,j)$ for input i generating output j is the probability $\zeta(\langle i,j\rangle)$ for the pair $\langle i,j\rangle$ to occur conditioned on the probability $\sum_{t=1}^{m} \zeta(\langle i,t\rangle)$ that input i is produced at all. Thus relation p satisfies the equation

$$\zeta(\langle i,j\rangle) = \left(\sum_{t=1}^{m} \zeta(\langle i,t\rangle)\right) \cdot p(i,j).$$

This is the discrete version of Prop. 1. In contrast to the discrete case, however, the version of the disintegration of ζ with respect to its projection usually cannot be computed explicitly in the general case. ◇

There is a rather helpful interplay between the projection of $\mu \otimes K$ to the second component and $K^{\bullet}(\mu)$ which will be exploited later on:

Observation 1 *If $\mu \in \mathbf{S}(X)$ is a sub-probability measure, and $K : X \rightsquigarrow Y$ is a stochastic relation, the equality $\pi^{\flat}_{X\times Y,Y}(\mu \otimes K) = K^{\bullet}(\mu)$ holds.*

3 Nondeterminism: The Fringe Relation

Probabilistic modelling is a special case of nondeterministic modelling: we do not only indicate possible outcomes but also assign a weight to them. Thus it comes as a natural construction that each stochastic relation defines a set-valued relation, at least on Polish spaces. This relation is defined now, and we will have

a look at the correspondence between both types of relations. We will assume in this section that the SB-spaces are endowed with a fixed Polish topology.

The support $\mathsf{supp}(\mu)$ of a probability measure $0 \neq \mu \in \mathbf{S}(X)$ is the set of all points $x \in X$ such that each open neighborhood U of x has positive measure. This set is the smallest closed set F with $\mu(F) = \mu(X)$, it is denoted by $\mathsf{supp}(\mu)$; for completeness, $\mathsf{supp}(0) := \emptyset$ is defined for the zero measure $0 \in \mathbf{S}(X)$.

We investigate the set valued map $x \mapsto \mathsf{supp}(K(x))$, when K is a transition probability from the Polish space X to the Polish space Y. This map is the relational counterpart to a stochastic relation, as we will see. It is clear that the map takes values in the set of all closed nonempty subsets of a Polish space, and that for any open subset U of Y the set

$$\mathsf{supp}(K(\cdot))^{-1}[U] = \{x \in X |\ K(x)(U) > 0\}$$

is a measurable subset of X.

As an aside, we look at supp from an algebraic point of view. Polish spaces with continuous maps form the category $\mathfrak{Pol}$. For the Polish space X the space of all its probabilities $\mathbf{S}(X)$ is also a Polish space. We denote by $\mathfrak{SubProb}$ the subcategory whose objects are all spaces $\mathbf{S}(X)$ when X ranges over Polish spaces. A morphism $K : \mathbf{S}(X) \to \mathbf{S}(Y)$ is a continuous map between $\mathbf{S}(X)$ and $\mathbf{S}(Y)$ when both spaces carry their weak topologies. Following a result due to Giry [11, Theorem 1], the functor $\mathbf{G}$ which assigns each Polish space its space of sub-probability measures is the functorial part a monad in $\mathfrak{Pol}$. Denote by $\mathbf{G}_f$ the composition of $\mathbf{G}$ with the forgetful functor $\mathfrak{SubProb} \to \mathfrak{Set}$, the latter denoting the category of sets with maps as morphisms.

Let $\mathbf{F}(X)$ be the space of all nonempty closed subsets for a Polish space X, endowed with the Vietoris topology. This topology has as a subbase the sets $\{F |\ F \subseteq U_1\} \cap \{F |\ F \cap U_2 \neq \emptyset\}$ for the open sets $U_1, U_2 \subseteq X$.

Here things are a bit more complicated than in the probabilistic setting: if X is a compact metric space, so is $\mathbf{F}(X)$ [13, 4.9.12, 4.9.13]; if X is a Polish space, then the compacta in $\mathbf{F}(X)$ form a Polish space under the Vietoris topology. From [13, 4.9.7] it may be deduced that X is a compact metric space provided $\mathbf{F}(X)$ is a Polish space. Anyway, denote by $\mathfrak{CL}$ the category which has $\mathbf{F}(X)$ for Polish X as objects. A morphism $\mathbf{F}(f) := f^\sharp : \mathbf{F}(X) \to \mathbf{F}(Y)$ is induced by the continuous map $f : X \to Y$ through the topological closure of the images under closed sets, hence $f^\sharp(A) := (f[A])^{\mathsf{cl}}$ is defined. Clearly, $f^\sharp$ is continuous in the Vietoris topology, since f is under the metric topology, and since $(g \circ f)^\sharp = g^\sharp \circ f^\sharp$, we see that $\mathbf{F} : \mathfrak{Pol} \to \mathfrak{CL}$ is a functor. The discussion above indicates that $\mathbf{F}$ is in general no monad in $\mathfrak{Pol}$ (it is, however, when $\mathfrak{Pol}$ is replaced by the category of all compact metric spaces). Consequently, it is not possible to relate both monads directly. A weaker result may be obtained, however.

Compose this functor with the forgetful functor $\mathfrak{CL} \to \mathfrak{Set}$ to obtain the functor $\mathbf{F}_f$.

Proposition 2. $\mathsf{supp} : \mathbf{G}_f \stackrel{\bullet}{\to} \mathbf{F}_f$ *is a natural transformation.*

Hence the map $x \mapsto \mathsf{supp}(K(x))$ relating a transition probability to a set of elements with positive probability is given by a natural transformation.

Leaving this side track, we define the fringe relation:

Definition 2. *Let $K : X \rightsquigarrow Y$ be a stochastic relation between the Polish spaces X and Y. The* fringe relation R_K associated with K *is defined as*

$$R_K := \{\langle x, y\rangle \in X \times Y | y \in \mathsf{supp}(K(x))\}.$$

Conversely, let $R \subseteq X \times Y$ be a set-theoretic relation, then a stochastic relation $K : X \rightsquigarrow Y$ is said to satisfy R (abbreviated by $R \models K$) iff $R_K = R$ holds, hence iff R is just the fringe of K.

Example 4. Let $f : X \to Y$ be a measurable map between the Polish spaces X and Y, and put $K(x) := \delta_{f(x)}$, δ_y denoting as usual the Dirac measure on y. Then $K : X \rightsquigarrow Y$ is a stochastic relation for which $R_K = Graph(f)$ holds. $\diamond$

Investigating the relationship between the stochastic relation K and its fringe R_K, we find that composition carries over as follows:

Observation 2 *Let $K : X \rightsquigarrow Y$ and $L : Y \rightsquigarrow Z$ be stochastic relations, then*

1. $R_L \circ R_K \subseteq R_{K;L}$,
2. *suppose that for each $x \in X$ the probability $K(x)(G)$ is positive for each open set $G \subseteq X$, then also $R_{K;L} \subseteq R_L \circ R_K$.*

$R \models K$ indicates that, if R is the nondeterministic specification of a system, stochastic relation K is its probabilistic refinement. Define for $K, K' : X \rightsquigarrow Y$, and for $0 \leq p \leq 1$ the stochastic relation $K \oplus_p K'$ upon defining

$$(K \oplus_p K')(x)(B) := p \cdot K(x)(B) + (1-p) \cdot K'(x)(B),$$

(thus $(K \oplus_p K')(x)$ is just the convex combination of the measures $K(x)$ and $K'(x)$). The operator $\oplus_p$ is interpreted as a weighted choice operator. It is easy to see that the following holds:

$$\frac{R \models K \;\; R \models K' \;\; 0 \leq p \leq 1}{R \models (K \oplus_p K')}$$

Consequently, the set of all stochastic relations satisfying a given nondeterministic specification is convex, hence closed under weighted choice. Convexity models the observation that nondeterministic systems are underspecified, as compared to stochastic ones (cf. the discussion in [15]).

Each stochastic relation has a fringe, and the inverse correspondence can be established under suitable topological assumptions: Given a set-valued relation R, a stochastic relation K that satisfies R can be found. For this, R has to take closed values, and a measurability condition is imposed:

Proposition 3. *Let $R \subseteq X \times Y$ a relation (X, Y Polish) such that*

1. $\forall x \in X : R(x) := \{y \in Y |\ \langle x, y\rangle \in R\} \in \mathbf{F}(Y)$,
2. *whenever $U \subseteq Y$ is open, $\{x \in X |\ R(x) \cap U \neq \emptyset\}$ is a measurable subset of X.*

If Y is σ-compact, or if $R(x)$ assumes compact values for each $x \in X$, then there exists a stochastic relation $K : X \rightsquigarrow Y$ with $R \models K$.

Thus each set-valued relation can be represented by a stochastic one under the conditions stated above, so that each nondeterministic specification can be satisfied by a stochastic relation.

4 Converse Relations

Given a sub-stochastic matrix $(p(i,j))_{1\leq i\leq n, 1\leq j\leq m}$ representing a stochastic relation $\{1,\ldots,n\} \rightsquigarrow \{1,\ldots,m\}$ and an initial distribution, the Introduction shows that the probability $p_\mu^\smile(j)(i)$ of responding with $j \in \{1,\ldots,m\}$ on a stimulus $i \in \{1,\ldots,n\}$ is calculated as

$$p_\mu^\smile(j)(i) = \frac{\mu(i)\cdot p(i,j)}{\sum_t \mu(t)\cdot p(t,j)}.$$

The probability $p_\mu^\smile$ under consideration reverses p given an initial distribution, so is regarded as the converse of p (*inverse* might at first sight be considered a better name, but this seems to suggest invertibility of the matrix associated with p).

In view of Examples 3 and 2, this amounts to the disintegration of $\mu \otimes p$ with respect to the distribution $p^\bullet(\mu) = \pi^\flat_{X\times Y,Y}(\mu \otimes p)$.

This observation guides the way for the definition of the converse for a general stochastic relation. Fix a stochastic relation $K : X \rightsquigarrow Y$, and a sub-probability measure $\mu \in \mathbf{S}(X)$. Then $\mu \otimes K \in \mathbf{S}(X \times Y)$ has a kind of natural converse: define $\tau := r^\flat(\mu \otimes K)$, where $r : X \times Y \to Y \times X$ switches components. Thus $r[R] = R^\smile := \{\langle y, x\rangle | \langle x, y\rangle \in R\}$, whenever $R \subseteq X \times Y$ is a relation, so r produces the converse. Because $\tau \in \mathbf{S}(Y \times X)$, this measure is — according to Prop. 1 — representable through a stochastic relation $K_\mu^\smile : Y \rightsquigarrow X$ by writing $\tau = \pi^\flat_Y(\tau) \otimes K_\mu^\smile$. Since $\pi^\flat_{Y\times X,Y}(\tau) = K^\bullet(\mu)$ by Obs. 1, the definition of the converse of a stochastic relation now reads as follows.

Definition 3. *The* μ-converse $K_\mu^\smile$ *of the stochastic relation K with respect to the input probability μ is defined by the equation $r^\flat(\mu \otimes K) = K^\bullet(\mu) \otimes K_\mu^\smile$.*

It is remarked that the converse $K_\mu^\smile$ always exists, and that it is unique μ-almost everywhere. Since

$$\mu(A) = (\mu \otimes K)(A \times Y) = (K^\bullet(\mu) \otimes K_\mu^\smile)((Y \times A)^\smile)$$

is true for the Borel set $A \subseteq X$,

$$\mu(A) = \int_X \int_Y K_\mu^\smile(A)\ K(x)(dy)\ \mu(dx) = \int_Y K_\mu^\smile(A)\ K^\bullet(\mu)(dy),$$

we infer that $\mu = \left(K_\mu^\smile\right)^\bullet(K^\bullet(\mu)) = (K;K_\mu^\smile)^\bullet(\mu)$ holds. Hence the converse $K_\mu^\smile$ solves the equation $\mu = (K;T)^\bullet(\mu)$ for T. This equation does, however, not determine the converse uniquely. This is so because it is an equation in terms of the Borel sets of X, hence may only be carried over to the "strip" $\{A \times Y |\ A \subseteq X \text{ Borel}\}$ on the product $X \times Y$. This is not enough to determine a measure on the entire product.

A *probabilistic interpretation* using regular conditional distributions may be given as follows: Let $(\Omega, \mathcal{A}, \mathbb{P})$ be a probability space, $\zeta_i : \Omega \to X_i$ random variables with values in the Polish spaces X_i (i = 1, 2). Let μ be the joint distribution of $\langle \zeta_1, \zeta_2 \rangle$, and let μ_i be the marginal distribution of ζ_i. If $\pi_i : X_1 \times X_2 \to X_i$ are the projections, then clearly $\mu_i = \pi_i(\mu)$. K denotes the regular conditional distribution of ζ_2 given ζ_1, thus we have for the Borel sets $A_i \subseteq X_i$

$$\begin{aligned}\mathbb{P}(\{\omega \in \Omega | \zeta_1(\omega) \in A_1, \zeta_2(\omega) \in A_2\}) &= \mu(A_1 \times A_2) \\ &= \int_{A_1} K(x_1)(A_2)\ \mu_1(dx_1).\end{aligned}$$

We will show now that $K_{\mu_1}^\smile$ is the regular conditional distribution of ζ_1 given ζ_2. In fact, let L be the latter distribution, then the definitions of K and L, resp., imply $K^\bullet(\mu_1) = \mu_2$ and $L^\bullet(\mu_2) = \mu_1$. Let $A_i \subseteq X_i$ be Borel sets, then $(K^\bullet(\mu_1) \otimes L)(A_2 \times A_1) = (\mu_1 \otimes K)(A_1 \times A_2)$.

Interpreting a stochastic relation as a regular conditional distribution of a random variable ζ_1 given ζ_2, its converse may be interpreted as the conditional distribution of ζ_2 given ζ_1. The start probability μ in the definition of $K_\mu^\smile$ is then interpreted as a marginal distribution. This is essentially the probabilistic setting for the definition of the converse in [1].

Returning to the general case, the defining equation for the converse is spelled out in terms of an integral (where $D^x := \{y \in Y | \langle y, x \rangle \in D\}$ for $D \subseteq Y \times X$, the cut D_y is defined above):

$$\int_X K(x)(D^x)\ \mu(dx) = \int_Y K_\mu^\smile(y)(D_y)\ K^\bullet(\mu)(dy).$$

This will be generalized and made use of later:

Observation 3 *Let $f \in \mathcal{M}(X \times Y)$, then this identity holds:*

$$\int_X \int_Y f(x,y)\ K(x)(dy)\ \mu(dx) = \int_Y \int_X f(x,y)\ K_\mu^\smile(y)(dx)\ K^\bullet(\mu)(dy).$$

Thus the order of integration of f may be interchanged, as in Fubini's Theorem, but, unlike that Theorem, we have to adjust the measures used for integration.

Some properties of forming the converse will be investigated now. We begin with an analogue of the property $R^{\smile\smile} = R$ which holds for the set theoretic converse. Taking the initial distribution into account, this property is very similar for the probabilistic case.

Proposition 4. *If $K : X \rightsquigarrow Y$, and if $\mu \in \mathbf{S}(X)$, then $(K_\mu^{\smile})^{\smile}_{K^\bullet(\mu)} = K$. holds everywhere except possibly on a set of μ-measure zero.*

The question under what condition a stochastic relation may be represented as the converse of another relation is a little more difficult to answer than for the set-valued case. In view of the probabilistic interpretation using conditional distributions, however, the following solution arises naturally.

Corollary 1. *Let $L : Y \rightsquigarrow X$ be a stochastic relation, and $\mu \in \mathbf{S}(X)$. Then these conditions are equivalent:*

1. *$\mu = L^\bullet(\nu)$ for some $\nu \in \mathbf{S}(Y)$,*
2. *$L = K_\mu^{\smile}$ for some $K : X \rightsquigarrow Y$.* □

Thus $L : Y \rightsquigarrow X$ may be written in a variety of ways as the converse of a stochastic relations, viz., $L = (K_\nu)^{\smile}_{L^\bullet(\nu)}$ for an arbitrary $\nu \in \mathbf{S}(Y)$ (where the relation $X \rightsquigarrow Y$ depends on ν). This is in marked contrast to the set-theoretic case, where the converse of the converse of a relation is the relation itself, hence unique.

Compatibility of composition and forming the converse is an important property in the world of set-theoretic relations. In that case it is well known that $(R;S)^{\smile} = S^{\smile};R^{\smile}$ always holds. The corresponding property for stochastic relations reads

Proposition 5. *Let $K : X \rightsquigarrow Y, L : Y \rightsquigarrow T$ be stochastic relations, and let $\mu \in \mathbf{S}(X)$ be an initial distribution. Then $(K;L)_\mu^{\smile} = L^{\smile}_{K^\bullet(\mu)};K_\mu^{\smile}$ holds.*

We see that there are some algebraic similarities between set-theoretic and stochastic relations. There are exceptions, though. Take e.g. *Schröder's Cycle Rule* $Q \circ R \subseteq S \Leftrightarrow Q^{\smile} \circ \overline{S} \subseteq \overline{R} \Leftrightarrow \overline{S} \circ R^{\smile} \subseteq \overline{Q}$, the bar denoting complementation ([18, 3.2 (xii)] or [4, Def. 3.1.1]). This rule is very helpful in practical applications, but it does not enjoy a direct counterpart for stochastic relations, since the respective notions of negation, and of containment do not carry over. —

If $\mu(A) = 0$ for some Borel set $A \subseteq X$, then $K_\mu^{\smile}(y)(A) = 0$ holds $K^\bullet(\mu)$-almost everywhere on Y (i.e., for all $y \in Y$ outside a set of $K^\bullet(\mu)$-measure zero). In fact, we can say more by scrutinizing the relationship between $K_\mu^{\smile}, K$ and μ. This leads to a rather surprising compactness result of the set of measures comprising the converse.

Recall that for $\mu, \nu \in \mathbf{S}(X)$ the measure ν is called *absolutely continuous* w. r. t. μ iff for every measurable set $A \subseteq X$ the implication $\mu(A) = 0 \Rightarrow \nu(A) = 0$ holds; this is indicated by $\nu \ll \mu$. It is well known [3, Sect. 32] that $\nu \ll \mu$ is equivalent to

$$\forall \varepsilon > 0 \; \exists \delta > 0 : \; [\mu(A) < \delta \Rightarrow \nu(A) < \varepsilon].$$

Absolute continuity is used for defining morphisms between probability spaces based on Polish spaces in [1, Def. 7.8] which in turn serves for defining the converse of a stochastic relation; we use it here for characterizing the measures comprising the converse. A subset $M \subseteq \mathbf{S}(X)$ is accordingly called *uniformly absolutely continuous* w.r.t. μ (indicated by $M \ll \mu$) iff given $\varepsilon > 0$ there exists $\delta > 0$ such that $\sup_{\nu \in M} \nu(A) < \varepsilon$ whenever $\mu(A) < \delta$ holds. It will be shown now that the set of measures constituting the converse is uniformly absolutely continuous except on a very small set:

Proposition 6. *Let $K : X \rightsquigarrow Y$ be a stochastic relation, and $\mu \in \mathbf{S}(X)$. Then for each version $K_\mu^\smile$ of the converse of K with respect to μ there exists a Borel set $A \subseteq Y$ for which $K(x)(A) = 0$ is true for μ-almost all $x \in X$ so that $\{K_\mu^\smile(y) | y \notin A\} \ll \mu$ holds.*

This implies that the set $\{K_\mu^\smile(y) | y \notin A\}$ is topologically not too large. Since we deal with a specific topology on the set of all sub-probability measures, we fix a Polish topology on the input space which in turn induces the topology of weak convergence on $\mathbf{S}(X)$.

Corollary 2. *Let X be a Polish space, endow $\mathbf{S}(X)$ with the topology of weak convergence, and let Y be an SB-space. Given $K : X \rightsquigarrow Y$ and $\mu \in \mathbf{S}(X)$, there exists a Borel set $A \subseteq Y$ with $K(x)(A) = 0$ for μ-almost all $x \in X$ so that the set $\{K_\mu^\smile(y) | y \notin A\}$ is a relatively compact subset of $\mathbf{S}(X)$.*

Finally, let us have a look at the fringe relation: it turns out that $(R_K)^\smile$ does not necessarily coincide with $R_{K_\mu^\smile}$.

Example 5. Define K as in Example 4, then $\mu \otimes K = id_X \times f^\flat(\mu)$ holds for $\mu \in \mathbf{P}(X)$, so that $\pi_Y^\flat(\mu \otimes K) = f^\flat(\mu)$ is inferred. For the Borel sets $A \subseteq X, B \subseteq Y$ the equalities

$$(\mu \otimes K)(A \times B) = \mu(A \cap f^{-1}[B]) = \left(f^\flat(\mu) \otimes K_\mu^\smile\right)(B \times A).$$

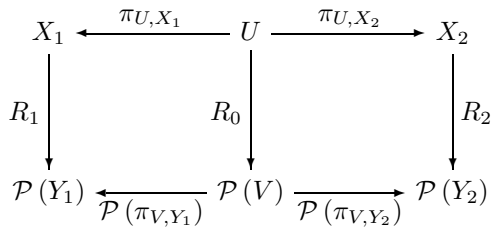
hold. Now put $\mu = \delta_{x'}$ for some $x' \in X$, then the constant relation $K_\mu^\smile(y) = \delta_{x'}$ is a version of the converse, hence

$$R_{K_\mu^\smile} = \{\langle y, x' \rangle | y \in Y\} \neq Graph(f)^\smile = (R_K{}^\smile).$$

Thus building the fringe relation and forming the converse does not commute. ◇

5 Bisimulations

Call the relations $R_1 \subseteq X_1 \times Y_1$ and $R_2 \subseteq X_2 \times Y_2$ bisimilar iff there exists $U \subseteq X_1 \times X_2$ and $V \subseteq Y_1 \times Y_2$ and a relation $R_0 \subseteq U \times V$ (U, V and R_0 are called *mediating*) such that this diagram is commutative:



Here $\mathcal{P}$ is the powerset functor, and relations are interpreted as set valued maps. This is the definition of bisimilarity for coalgebras [17, 2] adapted to the situation at hand.

Defining for a relation R the *yield* relation

$$x \vdash_R y \Longleftrightarrow \langle x, y\rangle \in R$$

in analogy to the transition relation $\rightarrow_R^a$ investigated for coalgebras, it is easy to see that R_1 and R_2 are bisimilar iff

1. for all $\langle x_1, x_2\rangle \in U$, if $x_1 \vdash_{R_1} y_1$, then there exists $y_2 \in Y_2$ such that $\langle y_1, y_2\rangle \in V$ and $x_2 \vdash_{R_2} y_2$,
2. for all $\langle x_1, x_2\rangle \in U$, if $x_2 \vdash_{R_2} y_2$, then there exists $y_1 \in Y_2$ such that $\langle y_1, y_2\rangle \in V$ and $x_1 \vdash_{R_1} y_1$.

In fact, Rutten's proof [17, Ex. 2.1] carries over. Bisimulations will be studied now for stochastic relations, and the goal is to show that bisimilar relations give rise to bisimilar converses. We first define bisimilarity for stochastic relations and show that under a mild condition bisimilarity is transitive. Then we establish that the operations we are working with, i.e., forming products of measures and relations, and transporting measures through relations, maintain bisimilarity. This holds also for disintegration, and having established this, a small step will be necessary to show that converses will respect bisimilarity.

Bisimulations are usually defined through spans of morphisms in a suitable category. In fact, a stochastic relation $K : X \rightsquigarrow Y$ can be considered as an object $\langle X, Y, K\rangle$ in the comma category $\mathbb{1}_{\mathcal{SB}} \downarrow \mathbf{S}$, where $\langle \alpha, \beta\rangle : \langle X, Y, K\rangle \rightarrow \langle X', Y', K'\rangle$ is a morphism iff $\alpha : X \rightarrow X'$ and $\beta : Y \rightarrow Y'$ are measurable such that $K \circ \alpha = \beta^\flat \circ K'$ holds. A *1-bisimulation* $\langle O, \Gamma_1, \Gamma_2\rangle$ for objects O_1 and O_2 is then an object O together with two morphisms $\Gamma_1 : O \rightarrow O_1$ and $\Gamma_2 : O \rightarrow O_2$. This notion of bisimilarity was discussed and investigated in [8] and specialized there to the present notion of bisimulation (called *2-bisimulation* in [8]), which is similar in spirit to the one given above for set valued relations:

Definition 4. *Let $K_1 : X_1 \rightsquigarrow Y_1$ and $K_2 : X_2 \rightsquigarrow Y_2$ be stochastic relations, where all participating spaces are SB-spaces. Then $N : U \rightsquigarrow V$ is called a* bisimulation for K_1 and K_2 *iff these conditions are satisfied:*

1. *$U \subseteq X_1 \times X_2$ and $V \subseteq Y_1 \times Y_2$ are SB-spaces,*
2. *$K_1 \circ \pi_{A,X_1} = \pi^\flat_{B,Y_1} \circ N$ and $K_2 \circ \pi_{A,X_2} = \pi^\flat_{B,Y_2} \circ N$ hold.*

Standard arguments show that N is a bisimulation for K_1 and K_2 iff (in the notation of Def. 4)

$$\int_{Y_i} f_i \; dK_i(x_i) = \int_V f_i \circ \pi_{V,Y_i} \; dN(x_1, x_2)$$

hold for each pair $\langle x_1, x_2 \rangle \in U$, and for each $f_i \in \mathcal{M}(Y_i)$ $(i = 1, 2)$. This condition is sometimes easier to handle.

Bisimulation turns out to be transitive under a rather mild condition of surjectivity. This property can be established using the existence of semi-pullbacks for stochastic relations (recall that a *semi-pullback* for a pair of morphisms $f_1 : a_1 \to c, f_2 : a_2 \to c$ in a category is a pair of morphisms $g_1 : b \to a_1, g_2 : b \to a_2$ with $f_1 \circ g_1 = f_2 \circ g_2$). The plan of attack is as follows: 1-bisimilarity is a transitive relation under the assumption of surjectivity [9, Theorem 2], and the comparison between 1-bisimilarity and bisimilarity from [8, Prop. 5] shows that both notions are equivalent under a condition of measurability. This technical condition which will be established here.

Proposition 7. *Let $K_i : X_i \rightsquigarrow Y_i$ $(i = 1, 2, 3)$ be stochastic relations, and assume that $N_1 : U_1 \rightsquigarrow V_1$ and $N_2 : U_2 \rightsquigarrow V_2$ are bisimulations for K_1, K_2 and K_2, K_3, resp. Assume that all projections are onto. Then there exists a bisimulation $N_3 : U_3 \rightsquigarrow V_3$ for K_1, K_3.*

In order to show that bisimilar relations give rise to bisimilar converses, it is practical to introduce the notion of bisimilarity for sub-probability measures, too; it is easy to see that the same notion of bisimilarity arises when one restricts oneself to constant stochastic relations.

Definition 5. *Let X_1, X_2 be SB-spaces with $\mu_i \in \mathbf{S}(X_i)$ $(i = 1, 2)$. Then $\langle X_1, \mu_1 \rangle$ is said to be* bisimilar to $\langle X_2, \mu_2 \rangle$ *iff there exists a subset $Z \subseteq X_1 \times X_2$ and $\zeta \in \mathbf{S}(Z)$ such that*

1. *Z is a SB-space,*
2. *$\mu_1 = \pi^\flat_{Z,X_1}(\zeta)$ and $\mu_2 = \pi^\flat_{Z,X_2}(\zeta)$.*

$\langle Z, \zeta \rangle$ is said to mediate between $\langle X_1, \mu_1 \rangle$ and $\langle X_2, \mu_2 \rangle$.

Bisimulations are maintained by forming products, and by transporting a measure through a stochastic relation, as we will see now:

Proposition 8. *Let $K_i : X_i \rightsquigarrow Y_i$ be bisimilar stochastic relations over the SB-spaces X_i, Y_i for $i = 1, 2$ such that $N : U \rightsquigarrow V$ mediates between them, and assume that $\mu_i \in \mathbf{S}(X_i)$ such that $\langle X_1, \mu_1 \rangle$ and $\langle X_2, \mu_2 \rangle$ are bisimilar with mediating $\langle Z, \zeta \rangle$. Assume that $Z \subseteq U$ holds. then*

1. *$\langle Y_1, {K_1}^\bullet(\mu_1) \rangle$ is bisimilar to $\langle Y_2, {K_2}^\bullet(\mu_2) \rangle$ with mediating $\langle V, N^\bullet(\zeta) \rangle$,*
2. *$\langle X_1 \times Y_1, \mu_1 \otimes K_1 \rangle$ is bisimilar to $\langle X_2 \times Y_2, \mu_2 \otimes K_2 \rangle$ with mediating $\langle t[E], t^\flat(\zeta \otimes N) \rangle$, where $E := Z \times V$ and $t(x_1, x_2, y_1, y_2) := \langle x_1, y_1, x_2, y_2 \rangle$.*

The argumentation above shows that bisimilar relations and bisimilar initial distributions lead to bisimilar measures on the product. The process can be reversed: the idea is that disintegrating bisimilar measures on a product leads to bisimilar stochastic relations.

Proposition 9. *Let X_i, Y_i be SB-spaces, $\mu_i \in \mathbf{S}(X_i \times Y_i)$ for $i = 1, 2$. Assume that $\langle X_1 \times Y_1, \mu_1 \rangle$ is bisimilar to $\langle X_2 \times Y_2, \mu_2 \rangle$. Define for $i = 1, 2$ the stochastic relations $K_i : X_i \rightsquigarrow Y_i$ as the disintegrations of μ_i w.r.t $\pi^{\flat}_{X_i \times Y_i, X_i}(\mu_i)$. Then K_1 is bisimilar to K_2.*

Showing that bisimilarity is maintained when forming the converse is now an easy consequence:

Corollary 3. *Under the assumptions of Prop. 8, $K^{\smile}_{1,\mu_1}$ is bisimilar to $K^{\smile}_{2,\mu_2}$.*

6 Related Work

The generalization of set-based relations to probabilistic ones appears straightforward: replace the nondeterminism inherent in these relations by randomness. Panangaden [14] carries out a very elegant construction, arguing as follows: the powerset functor is a monad which has relations as morphisms in its Kleisli category [12], the functor that assigns each measurable space the set of all (sub-) probability measures is also a monad having transition probabilities as morphisms in its Kleisli category [11]. This parallel justifies their characterization as probabilistic relations. The category **SRel** of measurable spaces with transition sub-probabilities is scrutinized closer in [14], and an application to Kozen's semantics of probabilistic programs is given. Stochastic relations are underlying stochastic automata; they were introduced and investigated in [7] as a generalization of finite stochastic machines. Abramsky, Blute, and Panangaden [1] investigate the category **PRel** of probability spaces, hereby introducing the converse of a probabilistic relation as we do through the product measure (Cor. 7.7). The process by which they arrive at this construction (Theorem 7.6) is quite similar to disintegration, as proposed here but makes heavier use of absolute continuity (in fact, morphisms in **PRel** use absolute continuity in a crucial way). The argumentation in the present paper seems to be closer to the set-theoretic case by looking at what happens when we compute the probability for a converse relation. Further investigations of the converse do not include the anti-commutative law. This is probably due to the fact that integration technique are directly used in the present paper (while [1] prefers arguing with absolute continuity, and consequently, with the Radon-Nikodym Theorem).

The notion of bisimilarity is — as in [8] — adapted from [6, 17] to the situation at hand. Transitivity of bisimulation is demonstrated in [10] for universally measurable stochastic relations case, but left open for the general case of Borel measurable transition probabilities; [9] gives a full solution to this problem.

The observation that each transition probability on a Polish space spawns a measurable set-valued function through the support function, hence a relation,

was used in [7] for investigating the relationship between nondeterministic and stochastic automata. It could be shown that each nondeterministic automaton can be represented through a stochastic one, and that this representation is preserved through the sequential work of the automata. Measurable selections play a major role, but the results are not formulated in terms of monads or categories.

7 Conclusion

Stochastic relations are generalizations of Markov processes. The converse of a stochastic relation is investigated, in particular it is shown that it satisfies some of the algebraic laws which rule their set-theoretic counterparts. Those relations that arise as converses are characterized, and it is shown that the set of all sub-probabilities comprising the converse is topologically quite small, i.e., is relatively compact in the weak topology. It is demonstrated that forming the converse does respect bisimulations — if the models one starts with are bisimilar, the converses will be, too. For a special case which includes the reals it is shown that bisimilarity of stochastic relations is a transitive relation; the proof makes use of the fact that semi-pullbacks exist in the corresponding category.

The nondeterminism inherent in a stochastic relation is identified, and it could be shown that nondeterministic and stochastic relations are related via a natural transformation that is induced by the support of finite measures. It is shown that the stochastic relations satisfying a nondeterministic one is convex, so that a nondeterministic specification provides a large degree of freedom for probabilistic satisfaction.

Further work will address the characterization of bisimilarity more closely in order to find necessary and sufficient conditions indicating under which two probabilistically related components are bisimilar.

References

[1] S. Abramsky, R. Blute, and P. Panangaden. Nuclear and trace ideal in tensored *-categories. *Journal of Pure and Applied Algebra*, 143(1 – 3):3 – 47, 1999.

[2] P. Aczel and N. Mendler. A final coalgebra theorem. In H. H. Pitt, A. Poigne, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 357 – 365, 1989.

[3] P. Billingsley. *Probability and Measure*. John Wiley and Sons, New York, 3 edition, 1995.

[4] C. Brink, W. Kahl, and G. Schmidt, editors. *Relational Methods in Computer Science*. Advances in Computing Science. Springer-Verlag, Wien, New York, 1997.

[5] D. Cantone, E. G. Omodeo, and A. Policriti. *Set Theory for Computing*. Springer-Verlag, 2001. In print.

[6] J. Desharnais, A. Edalat, and P. Panangaden. Bisimulation of labelled Markov-processes. Technical report, School of Computer Science, McGill University, Montreal, 1998.

[7] E.-E. Doberkat. *Stochastic Automata — Nondeterminism, Stability, and Prediction*, volume 113 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

[8] E.-E. Doberkat. The demonic product of probabilistic relations. In Mogens Nielsen and Uffe Engberg, editors, *Proc. Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 113 – 127, Berlin, 2002. Springer-Verlag.

[9] E.-E. Doberkat. Semi-pullbacks and bisimulations in categories of stochastic relations. Technical Report 130, Chair for Software Technology, University of Dortmund, November 2002.

[10] A. Edalat. Semi-pullbacks and bisimulation in categories of Markov processes. *Math. Struct. in Comp. Science*, 9(5):523 – 543, 1999.

[11] M. Giry. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes in Mathematics*, pages 68 – 85, Berlin, 1981. Springer-Verlag.

[12] S. Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, Berlin, 2 edition, 1997.

[13] E. Michael. Topologies on spaces of subsets. *Trans. Am. Math. Soc.*, 71(2):152 – 182, 1951.

[14] P. Panangaden. Probabilistic relations. In C. Baier, M. Huth, M. Kwiatkowska, and M. Ryan, editors, *Proc. PROBMIV*, pages 59 – 74, 1998. Also available from the School of Computer Science, McGill University, Montreal.

[15] P. Panangaden. Does combining nondeterminism and probability make sense? *Bulletin of the EATCS*, (75):182 – 189, Oct. 2001.

[16] K. R. Parthasarathy. *Probability Measures on Metric Spaces*. Academic Press, New York, 1967.

[17] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3 – 80, 2000. Special issue on modern algebra and its applications.

[18] A. Tarski and S. Givant. *A Formalization of Set-Theory Without Variables*. Number 42 in Colloquium Publications. American Mathematical Society, Providence, R. I., 1987.

Type Assignment for Intersections and Unions in Call-by-Value Languages

Joshua Dunfield and Frank Pfenning

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{joshuad,fp}@cs.cmu.edu

Abstract. We develop a system of type assignment with intersection types, union types, indexed types, and universal and existential dependent types that is sound in a call-by-value functional language. The combination of logical and computational principles underlying our formulation naturally leads to the central idea of type-checking subterms in evaluation order. We thereby provide a uniform generalization and explanation of several earlier isolated systems. The proof of progress and type preservation, usually formulated for closed terms only, relies on a notion of definite substitution.

1 Introduction

Conventional static type systems are tied directly to the expression constructs available in a language. For example, functions are classified by function types $A \to B$, pairs are classified by product types $A * B$, and so forth. In more advanced type systems we find type constructs that are independent of any particular expression construct. The best-known examples are parametric polymorphism $\forall t.A$ and intersection polymorphism $A \wedge B$. Such types can be seen as expressing more complex properties of programs. For example, if we read the judgment $e : A$ as *e satisfies property A*, then $e : A \wedge B$ expresses that e satisfies both property A and property B. We call such types *property types*. Our long-term goal is to integrate a rich system of property types into practical languages such as Standard ML [9], in order to express and verify detailed invariants of programs as part of type-checking.

In this paper we design a system of property types specifically for call-by-value languages. We show that the resulting system is type-safe, that is, satisfies the type preservation and progress theorems. We include indexed types $\delta(i)$, intersection types $A \wedge B$, a greatest type $\top$, universal dependent types $\Pi a{:}\gamma.\, A$, union types $A \vee B$, an empty type $\bot$, and existential dependent types $\Sigma a{:}\gamma.\, A$. We thereby combine, unify, and extend prior work on intersection types [6], union types [11,2] and dependent types [17].

Several principles emerge from our investigation. Perhaps most important is that type assignment may visit subterms in evaluation order, rather than

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 250–266, 2003.

just relying on immediate subterms. We also confirm the critical importance of a logically motivated design for subtyping and type assignment. The resulting orthogonality of various property type constructs greatly simplifies the theory and allows one to understand each concept in isolation. As a consequence, simple types, intersection types [6], and indexed and dependent types [17] are extended *conservatively*. There are also interesting technical aspects in our proof of progress and preservation: Usually these can be formulated entirely for closed expressions; here we needed to generalize the properties by allowing so-called *definite substitutions*. Our type system is designed to allow effects (in particular, mutable references), but in order to concentrate on more basic issues, we do not include them explicitly in this paper (see [6] for the applicable techniques to handle mutable references).

The results in this paper constitute the first step towards a practical type system. The system of pure type assignment presented here is undecidable; to remedy this, we have formulated another version based on bidirectional type-checking (in the style of [6,7]) of programs containing some type annotations, where we variously *check* an expression against a type or else *synthesize* the expression's type. However, we do not yet have a formal proof of decidability, nor any significant experience with checking realistic programs. Our confidence in the practicality of the system rests on prior work on intersection and dependent types in isolation.

The remainder of the paper is organized as follows. We start by defining a small and conventional functional language with subtyping, in a standard call-by-value semantics. We then add several forms of property types: intersection types, indexed types, and universal dependent types. As we do so, we motivate our typing and subtyping rules through examples, showing how our particular formulation arises out of our demand that the theorems of *type preservation* and *progress* hold. Then we add the *indefinite* property types: the empty type $\bot$, the union type $\vee$, and the existential dependent type Σ. To be sound, these must visit subterms in evaluation order. After proving some novel properties of judgments and substitutions, we prove preservation and progress. Finally, we discuss related work and conclude.

2 The Base Language

We start by defining a standard call-by-value functional language (Figure 1) with functions, a unit type (used in a few examples), and recursion, to which we will add various constructs and types. Expressions do not contain types, because we are formulating a pure type assignment system. We distinguish between variables x that stand for values and variables f that stand for expressions, where the fs arise only from fixed points. The form of the typing judgment is $\Gamma \vdash e : A$ where Γ is a context typing variables x and f. The typing rules here are standard (Figure 2); the subsumption rule utilizes a subtyping judgment $\Gamma \vdash A \leq B$ meaning that A is a subtype of B in context Γ. The interpretation is that the set of values of type A is a subset of the set of values of type B. The context

$$A, B, C, D ::= \mathbf{1} \mid A \to B$$
$$e ::= x \mid f \mid () \mid \lambda x.\, e \mid e_1(e_2) \mid \mathbf{fix}\ f.\, e$$

Fig. 1. Syntax of types and terms in the initial language

$$\frac{\Gamma \vdash B_1 \leq A_1 \quad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2}\ (\to) \qquad \frac{}{\Gamma \vdash \mathbf{1} \leq \mathbf{1}}\ (\mathbf{1})$$
$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ (\text{var}) \qquad \frac{\Gamma(f) = A}{\Gamma \vdash f : A}\ (\text{fixvar}) \qquad \frac{\Gamma \vdash e : A \quad \Gamma \vdash A \leq B}{\Gamma \vdash e : B}\ (\text{sub})$$
$$\frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1(e_2) : B}\ (\to E) \qquad \frac{\Gamma, f{:}A \vdash e : A}{\Gamma \vdash \mathbf{fix}\ f.\, e : A}\ (\mathbf{fix})$$
$$\frac{\Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B}\ (\to I) \qquad \frac{}{\Gamma \vdash () : \mathbf{1}}\ (\mathbf{1}I)$$

Fig. 2. Subtyping and typing in the initial language

Γ is not used in the subtyping rules of Figure 2, but we subsequently augment the subtyping system with rules that refer to Γ. The rule $(\to)$ is the standard subtyping rule for function types, contravariant in the argument and covariant in the result; $(\mathbf{1})$ is obvious. It is easy to prove that subtyping is decidable, reflexive ($\Gamma \vdash A \leq A$), and transitive (if $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq C$ then $\Gamma \vdash A \leq C$); as we add rules to the subtyping system, we maintain these properties.

A call-by-value operational semantics defining a relation $e \mapsto e'$ is given in Figure 3. We use v for values, and write e value if e is a value. We write E for an evaluation context—a term containing a hole $[]$; $E[e']$ denotes E with its hole replaced by e'.

3 Definite Property Types

Definite types accumulate positive information about expressions. For instance, the intersection type $A \wedge B$ expresses the conjunction of the properties A and B. We later introduce *indefinite types* such as $A \vee B$ which encompass expressions that have either property A or property B, although it is unknown which one.

3.1 Refined Datatypes

We now add datatypes with refinements (Figure 4). $c(e)$ denotes a datatype constructor c applied to an argument e; the destructor **case** e **of** ms denotes a case over e with one layer of non-redundant and exhaustive matches ms. We also add pairs so a constructor can take exactly one argument, but elide the straightforward syntax and rules. Each datatype is *refined*, in the manner of [5], by an *atomic subtyping* relation $\preceq$ over *datasorts* δ. Each datasort identifies a subset of values of the form $c(v)$, yielding definite information about a value.

$$\begin{array}{rl} \text{Values} & v ::= x \mid () \mid \lambda x.\, e \\ \text{Evaluation contexts} & E ::= [] \mid E(e) \mid v(E) \end{array}$$

$$\frac{e' \;\mapsto_R\; e''}{E[e'] \;\mapsto\; E[e'']}\ (\text{ev-context}) \qquad \begin{array}{l} (\lambda x.\, e)\, v \;\mapsto_R\; [v/x]\, e \\ \mathbf{fix}\ f.\, e \;\mapsto_R\; [\mathbf{fix}\ f.\, e\ /\ f]\, e \end{array}$$

Fig. 3. A small-step call-by-value semantics

$$\begin{array}{rcl} ms & ::= & \cdot \mid c(x) \Rightarrow e \,|\, ms \\ e & ::= & \ldots \mid c(e) \mid \mathbf{case}\ e\ \mathbf{of}\ ms \\ v & ::= & \ldots \mid c(v) \\ E & ::= & \ldots \mid c(E) \mid \mathbf{case}\ E\ \mathbf{of}\ ms \end{array}$$

Fig. 4. Extending the language with datatypes

For example, datasorts true and false identify singleton subsets of values of the type bool.

A new subtyping rule defines subtyping for datasorts in terms of the atomic subtyping relation $\preceq$:

$$\frac{\delta_1 \preceq \delta_2}{\Gamma \vdash \delta_1 \leq \delta_2}\ (\delta)$$

To maintain reflexivity and transitivity of subtyping, we require the same properties of atomic subtyping: $\preceq$ must be reflexive and transitive.

Since we will subsequently further refine our datatypes by indices, we defer discussion of the typing rules.

3.2 Intersections

The typing $e : A \wedge B$ expresses that e has type A and type B. The subtyping rules for $\wedge$ capture this:

$$\frac{\Gamma \vdash A \leq B_1 \quad \Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \wedge B_2}\ (\wedge\mathrm{R}) \qquad \frac{\Gamma \vdash A_1 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\ (\wedge\mathrm{L}_1) \qquad \frac{\Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\ (\wedge\mathrm{L}_2)$$

We omit the common distributivity rule

$$\overline{(A \to B) \wedge (A \to B') \leq A \to (B \wedge B')}$$

which Davies and Pfenning showed to be unsound in the presence of mutable references [6]. Moreover, without the above rule, no subtyping rule contains more than one type constructor: the rules are orthogonal. As we add type constructors and subtyping rules, we will maintain this orthogonality.

On the level of typing, we can introduce an intersection with the rule

$$\frac{\Gamma \vdash v : A_1 \quad \Gamma \vdash v : A_2}{\Gamma \vdash v : A_1 \wedge A_2}\ (\wedge I)$$

$$
\begin{array}{rcl}
P & ::= & \bot \mid i \doteq j \mid \ldots \\
\Gamma & ::= & \cdot \mid \Gamma, x{:}A \mid \Gamma, a{:}\gamma \mid \Gamma, P
\end{array}
\qquad\qquad
\begin{array}{rcl}
\overline{\cdot} & = & \cdot \\
\overline{\Gamma, x{:}A} & = & \overline{\Gamma} \\
\overline{\Gamma, a{:}\gamma} & = & \overline{\Gamma}, a{:}\gamma \\
\overline{\Gamma, P} & = & \overline{\Gamma}, P
\end{array}
$$

Fig. 5. Propositions P, contexts Γ, and the restriction function $\overline{\Gamma}$

and eliminate it with

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_1}\ (\wedge E_1) \qquad \frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_2}\ (\wedge E_2)$$

Note that $(\wedge I)$ can only type values v, not arbitrary expressions, following Davies and Pfenning [6] who showed that in the presence of mutable references, allowing non-values destroys type preservation.

The $\wedge$-elimination rules are derivable via (sub) with the ($\wedge$L1) and ($\wedge$L2) subtyping rules. However, we include them because they are not derivable in a bidirectional system such as that of [7].

3.3 Greatest Type: $\top$

It is easy to incorporate a greatest type $\top$, which can be thought of as the 0-ary form of $\wedge$. The rules are simply

$$\frac{}{\Gamma \vdash A \leq \top}\ (\top\mathrm{R}) \qquad\qquad \frac{}{\Gamma \vdash v : \top}\ (\top I)$$

There is no left subtyping rule. The typing rule is essentially the 0-ary version of $(\wedge I)$, the rule for binary intersection. If we allow $(\top I)$ to type non-values, the progress theorem fails: $\vdash$ `()()` : $\top$, but `()()` is neither a value nor a redex.

3.4 Index Refinements and Universal Dependent Types Π

Now we add index refinements, which are dependent types over a restricted domain, closely following Xi and Pfenning [17], Xi [15,16], and Dunfield [7]. This refines datatypes not only by datasorts, but by indices drawn from some constraint domain: the type $\delta(i)$ is the refinement by δ and index i.

To accommodate index refinements, several changes must be made to the systems we have constructed so far. The most drastic is that Γ can include *index variables* a, b and propositions P as well as program variables. Because the program variables are irrelevant to the index domain, we can define a *restriction function* $\overline{\Gamma}$ that yields its argument Γ without program variable typings (Figure 5). No variable may appear twice in Γ, but ordering of the variables is now significant because of dependencies.

Our formulation, like Xi's, requires only a few properties of the constraint domain: There must be a way to decide a consequence relation $\overline{\Gamma} \models P$ whose interpretation is that given the index variable typings and propositions in $\overline{\Gamma}$, the proposition P must hold. There must be a relation $i \doteq j$ denoting index

equality. There must be a way to decide a relation $\overline{\Gamma} \vdash i : \gamma$ whose interpretation is that i has *sort* γ in $\overline{\Gamma}$. Note the stratification: terms have types, indices have sorts; terms and indices are distinct. Our proofs require that $\models$ be a consequence relation, that $\doteq$ be an equivalence relation, that $\cdot \not\models \bot$, and that both $\models$ and $\vdash$ have the obvious substitution and weakening properties; see [7] for details.

Each datatype has an associated atomic subtyping relation on datasorts, and an associated sort whose indices refine the datatype. In our examples, we work in a domain of integers $\mathcal{N}$ with $\doteq$ and some standard arithmetic operations ($+$, $-$, $*$, $<$, and so on); each datatype is refined by indices of sort $\mathcal{N}$. Then $\overline{\Gamma} \models P$ is decidable provided the inequalities in P are linear.

We add an infinitary definite type $\Pi a{:}\gamma.\, A$, introducing an index variable a universally quantified over indices of sort γ. One can also view Π as a dependent product restricted to indices (instead of arbitrary terms).

Example. Assume we define an datatype of integer lists: a list is either $\mathsf{Nil}()$ or $\mathsf{Cons}(h, t)$ for some integer h and list t. Refine this type by a datasort odd if the list's length is odd, even if it is even. We also refine the lists by their length, so Nil has type $\mathbf{1} \to \mathsf{even}(0)$, and Cons has type $(\Pi a{:}\mathcal{N}.\, \mathsf{int} * \mathsf{even}(a) \to \mathsf{odd}(a+1)) \wedge (\Pi a{:}\mathcal{N}.\, \mathsf{int} * \mathsf{odd}(a) \to \mathsf{even}(a+1))$. Then the function

$$\mathbf{fix}\ \mathit{repeat}.\ \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathsf{Nil} \Rightarrow \mathsf{Nil}\ |\ \mathsf{Cons}(h,t) \Rightarrow \mathsf{Cons}(h, \mathsf{Cons}(h, \mathit{repeat}(t)))$$

has type $\Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \mathsf{even}(2 * a)$.

To handle the indices, we modify the subtyping rule δ from Section 3.1 so that it checks (separately) the datasorts δ_1, δ_2 and the indices i, j:

$$\frac{\delta_1 \preceq \delta_2 \quad \overline{\Gamma} \vdash i \doteq j}{\Gamma \vdash \delta_1(i) \leq \delta_2(j)}\ (\delta)$$

We assume the constructors c are typed by a judgment $\overline{\Gamma} \vdash c : A \to \delta(i)$ where A is any type and $\delta(i)$ is some refined type. The typing $A \to \delta(i)$ need not be unique; indeed, a constructor should often have more than one refined type. The rule for constructor application is

$$\frac{\overline{\Gamma} \vdash c : A \to \delta(i) \quad \Gamma \vdash e : A}{\Gamma \vdash c(e) : \delta(i)}\ (\delta I)$$

To type **case** e **of** ms, we check that all the matches in ms have the same type, under a context appropriate to each arm; this is how propositions P arise. The context Γ may be contradictory ($\overline{\Gamma} \models \bot$) if the case arm can be shown to be unreachable by virtue of the index refinements of the constructor type and the value cased upon. In order to not typecheck unreachable arms, we have

$$\frac{\overline{\Gamma} \models \bot}{\Gamma \vdash e : A}\ (\text{contra})$$

We also do not check case arms that are unreachable by virtue of the *datasort* refinements. For a complete accounting of constructor typing and the rules for typing **case** expressions, see [7].

The subtyping rules for Π are

$$\frac{\Gamma \vdash [i/a]A \leq B \quad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash \Pi a{:}\gamma.\, A \leq B}\ (\Pi\mathrm{L}) \qquad \frac{\Gamma, b{:}\gamma \vdash A \leq B}{\Gamma \vdash A \leq \Pi b{:}\gamma.\, B}\ (\Pi\mathrm{R})$$

The left rule allows one to instantiate a quantified index variable a to an index i of appropriate sort. The right rule states that if $A \leq B$ regardless of an index variable b, A is also a subtype of $\Pi b{:}\gamma.\, B$. Of course, b cannot occur free in A.

The typing rules for Π are

$$\frac{\Gamma, a{:}\gamma \vdash v : A}{\Gamma \vdash v : \Pi a{:}\gamma.\, A}\ (\Pi I) \qquad \frac{\Gamma \vdash e : \Pi a{:}\gamma.\, A \quad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e : [i/a]\, A}\ (\Pi E)$$

Like $(\wedge I)$, and for similar reasons (to maintain type preservation), (ΠI) is restricted to values. Moreover, if γ is an empty sort, progress would fail if the rule were not thus restricted.

4 Indefinite Property Types

We now have a system with definite types $\wedge$, $\top$, Π; see [7] for a detailed account of this system and its bidirectional version. The typing and subtyping rules are both orthogonal and internally regular: no rule mentions both $\top$ and $\wedge$, $(\top I)$ is a 0-ary version of $(\wedge I)$, and so on. However, one cannot express the types of functions with indeterminate result type. A simple example is a `filter` f l function on lists of integers, which returns the elements of l for which f returns `true`. It has the ordinary type `filter` : (int→bool) → list → list. Indexing lists by their length, the refined type should look like

$$\texttt{filter} : \Pi n{:}\mathcal{N}.\, (\mathsf{int}{\to}\mathsf{bool}) \to \mathsf{list}(n) \to \mathsf{list}(\_)$$

But we cannot fill in the blank. Xi's solution [17,15] was to add dependent sums $\Sigma a{:}\gamma.\, A$ quantifying existentially over index variables. Then we can express the fact that `filter` returns a list of some indefinite length m as follows[1]:

$$\texttt{filter} : \Pi n{:}\mathcal{N}.\, (\mathsf{int}{\to}\mathsf{bool}) \to \mathsf{list}(n) \to (\Sigma m{:}\mathcal{N}.\, \mathsf{list}(m))$$

For similar reasons, we also occasionally need 0-ary and binary indefinite types—the empty type and union types, respectively. We begin with the binary case.

4.1 Unions

On values, the binary indefinite type should simply be a union in the ordinary sense: if $\vdash v : A \vee B$ then either $\vdash v : A$ or $\vdash v : B$. This leads to the following subtyping rules which are dual to the intersection rules.

$$\frac{\Gamma \vdash A_1 \leq B \quad \Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \vee A_2 \leq B}\ (\vee\mathrm{L}) \quad \frac{\Gamma \vdash A \leq B_1}{\Gamma \vdash A \leq B_1 \vee B_2}\ (\vee\mathrm{R}_1) \quad \frac{\Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \vee B_2}\ (\vee\mathrm{R}_2)$$

[1] The additional constraint $m \leq n$ can be expressed by a *subset sort*; see Xi [16,15].

The introduction rules directly express the simple logical interpretation:

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e : A \vee B}\ (\vee I_1) \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash e : A \vee B}\ (\vee I_2)$$

The elimination rule is harder to formulate. It is clear that if $e : A \vee B$ and e evaluates to a value v, then either $v : A$ or $v : B$. So we should be able to reason by cases, similar to the usual disjunction elimination rule in natural deduction. However, there are several complications. The first is that $A \vee B$ is a property type. That is, we cannot have a `case` construct in the ordinary sense since the members of the union are not tagged.[2]

As a simple example, consider

$$\begin{array}{l} f : (B \to D) \wedge (C \to D) \\ g : A \to (B \vee C) \\ x : A \end{array}$$

Then $f(g(x))$ should be type correct and have type D. At first this might seem doubtful, because the type of f does not directly show how to treat an argument of type $B \vee C$. However, whatever g returns must be a closed value v, and must therefore either have type B or type C. In both cases $f(v)$ should be well-typed and return a result of type D.

Note that we can distinguish cases on the result of $g(x)$ because it is evaluated *before* f is called.[3] In general, we allow case distinction on the type of the next expression to be evaluated. This guarantees both progress and preservation. The rule is then

$$\frac{\Gamma \vdash e' : A \vee B \quad \begin{array}{c} \Gamma, x{:}A \vdash E[x] : C \\ \Gamma, y{:}B \vdash E[y] : C \end{array}}{\Gamma \vdash E[e'] : C}\ (\vee E)$$

The use of the evaluation context $E[\,]$ guarantees that e' is the next expression to be evaluated, following our informal reasoning above. In the example, $e' = g(x)$ and $E[\,] = f[\,]$.

Several generalizations of this rule come to mind that are in fact unsound in our setting. For example, allowing simultaneous abstraction over several occurrences of e', as in a rule proposed in [2],

$$\frac{\Gamma \vdash e' : A \vee B \quad \begin{array}{c} \Gamma, x{:}A \vdash e : C \\ \Gamma, x{:}B \vdash e : C \end{array}}{\Gamma \vdash [e'/x]\, e : C}\ (\vee E')$$

is unsound here: two occurrences of the identical e' could return different results (the first of type A, the second of type B), while the rule above assumes consistency. Similarly, we cannot allow the occurrence of e' to be in a position where

[2] Pierce's `case` [11] is a syntactic marker for where to apply the elimination rule. Clearly, a pure type assignment system should avoid this. It appears we can avoid it even in a bidirectional system; further discussion is beyond the scope of this paper.

[3] If arguments were passed by name instead of by value, this would be unsound in a language with effects: evaluation of the same expression $e : A \vee B$ could sometimes return a value of type A and sometimes a value of type B.

it might not be evaluated. That is, in $(\vee E')$ it is not enough to require that there be exactly one occurrence of x in e, because, for example, if we consider the context

$$f : ((1 \to B) \to D) \wedge ((1 \to C) \to D),$$
$$g : A \to (B \vee C),$$
$$x : A$$

and term $f\,(\lambda y.\, g(x))$, then f may use its argument at multiple types, eventually evaluating $g(x)$ multiple times with different possible answers. Thus, treating it as if all occurrences must all have type B or all have type C is unsound. If we restrict the rule so that e' must be a value, as in [13], we obtain a sound but impractical rule—a typechecker would have to guess e', and if it occurs more than once, a subset of its occurrences.

A final generalization suggests itself: we might allow the subterm e' to occur exactly once, and in any position where it would definitely have to be evaluated exactly once for the whole expression to be evaluated. Besides the difficulty of characterizing such positions, even this apparently innocuous generalization is unsound for the empty type $\bot$.

4.2 The Empty Type

The 0-ary indefinite type is the empty or void type $\bot$; it has no values. For $\top$ we had one right subtyping rule; for $\bot$, following the principle of duality, we have one left rule:

$$\frac{}{\Gamma \vdash \bot \leq A}\ (\bot\mathrm{L})$$

For example, the term $\omega = (\mathbf{fix}\ f.\lambda x.\, f(x))\,()$ has type $\bot$. For an elimination rule $(\bot E)$, we can proceed by analogy with $(\vee E)$:

$$\frac{\Gamma \vdash e' : \bot}{\Gamma \vdash E[e'] : C}\ (\bot E)$$

As before, the expression typed must be an evaluation context E with redex e'. Viewing $\bot$ as a 0-ary union, we had two additional premises in $(\vee E)$, so we have none now. $(\bot E)$ is sound, but the generalization mentioned at the end of the previous section violates progress (Theorem 3). This is easy to see through the counterexample $(()\,())(\omega)$.

4.3 Existential Dependent Types: Σ

Now we add an infinitary indefinite type Σ. Just as we have come to expect, the subtyping rules are dual to the rules for the corresponding definite type (in this case Π):

$$\frac{\Gamma, a{:}\gamma \vdash A \leq B}{\Gamma \vdash \Sigma a{:}\gamma.\, A \leq B}\ (\Sigma\mathrm{L}) \quad \frac{\Gamma \vdash A \leq [i/b]\,B \quad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash A \leq \Sigma b{:}\gamma.\, B}\ (\Sigma\mathrm{R})$$

The typing rule that introduces Σ is simply

$$\frac{\Gamma \vdash e : [i/a]\, A \quad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e : \Sigma a{:}\gamma.\, A}\ (\Sigma I)$$

For the elimination rule, we continue with a restriction to evaluation contexts:

$$\frac{\Gamma \vdash e' : \Sigma a{:}\gamma.\, A \quad \Gamma, a{:}\gamma, x{:}A \vdash E[x] : C}{\Gamma \vdash E[e'] : C}\ (\Sigma E)$$

Not only is the restriction consistent with the elimination rules for $\bot$ and $\vee$, but it is required. The counterexample for $\bot$ suffices: Suppose that the rule were unrestricted, so that it typed any e containing some subterm e'. Let $e' = \omega$ and $e = (()())(\omega)$. Since $e' : \bot$, by subsumption e' has type $\Sigma a{:}\bot.\, A$ for any A, and by the (contra) rule, $a{:}\bot, x{:}A \vdash (()())x : C$ (where $\bot$ is the empty sort). Now we can apply the unrestricted rule to conclude $\vdash (()())e' : C$ for any C, contrary to progress.

4.4 Type-Checking in Evaluation Order

The following rule internalizes a kind of substitution principle for evaluation contexts and allows us to type-check a term in evaluation order.

$$\frac{\Gamma \vdash e' : A \quad \Gamma, x{:}A \vdash E[x] : C}{\Gamma \vdash E[e'] : C}\ (\text{direct})$$

Perhaps surprisingly, this rule is not only admissible but derivable in our system: from $e' : A$ we can conclude $e' : A \vee A$ and then apply $(\vee E)$. However, the corresponding bidirectional rule is not admissible, and so must be primitive in a bidirectional system [7].

Thus, in either the type assignment or bidirectional systems, we can choose to type-check the term in evaluation order. This has a clear parallel in Xi's work [15], which is bidirectional and contains both Π and Σ. There, the order in which terms are typed is traditional, not guided by evaluation order. However, Xi's elaboration algorithm in the presence of Π and Σ transforms the term into a let-normal form, which has a similar effect.

5 Properties of Subtyping

The rules of subtyping were formulated so that the premises are always smaller than the conclusion. Since we assume that $\models$ and $\vdash$ in the constraint domain are decidable, we obtain decidability immediately.

Theorem 1. *$\Gamma \vdash A \leq B$ is decidable.*

$$\frac{}{\Gamma' \vdash \cdot : \cdot}\ (\text{empty-}\sigma) \qquad \frac{\Gamma' \vdash \sigma : \Gamma \quad \overline{\Gamma'} \models [\sigma]P}{\Gamma' \vdash \sigma : \Gamma, P}\ (\text{prop-}\sigma)$$

$$\frac{\Gamma' \vdash \sigma : \Gamma \quad \overline{\Gamma'} \vdash i : \gamma}{\Gamma' \vdash \sigma, i/a : \Gamma, a{:}\gamma}\ (\text{ivar-}\sigma) \qquad \frac{\Gamma' \vdash \sigma : \Gamma \quad \Gamma' \vdash v : [\sigma]A}{\Gamma' \vdash \sigma, v/x : \Gamma, x{:}A}\ (\text{pvar-}\sigma)$$

Fig. 6. Substitution typing

We omitted rules for reflexivity and transitivity of subtyping without loss of expressive power, because they are admissible.

Lemma 1 (Reflexivity and Transitivity of $\leq$). *For any context Γ, $\Gamma \vdash A \leq A$. If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq C$ then $\Gamma \vdash A \leq C$.*

Proof. For reflexivity, by induction on A. For transitivity, by induction on the derivations; in each case at least one derivation becomes smaller. In the cases (ΣR)/(ΣL) and (ΠR)/(ΠL) we substitute an index i for a parameter a in a derivation. □

In addition we have a large set of inversion properties, which are purely syntactic in our system. We elide the lengthy statement of these properties here.

6 Properties of Values

For the proof of type safety, we need a key property: *values are always definite*. That is, once we obtain a value v, even though v might have type $A \vee B$, it *must* be possible to assign a definite type to v. In order to make this precise, we formulate substitutions σ that substitute values and indices, respectively, for several program variables x and index variables a. First we prove a simple lemma relating values and evaluation contexts.

Lemma 2 (Value monotonicity). *If $E[e']$ value then: (1) e' value; (2) for any v value, $E[v]$ value.*

Proof. By structural induction on E. □

6.1 Substitutions

Figure 6 defines a typing judgment for substitutions $\Gamma' \vdash \sigma : \Gamma$. It could be more general; here we are only interested in substitutions of values for program variables and indices for index variables that verify the logical assumptions of the constraint domain. Note in particular that substitutions σ do *not* substitute for fixed point variables. Application of a substitution σ to a term e or type A, is in the usual (capture-avoiding) manner.

Lemma 3 (Substitution).

(i) *If* $\Gamma \vdash A \leq B$ *and* $\Gamma' \vdash \sigma : \Gamma$, *then* $\Gamma' \vdash [\sigma]A \leq [\sigma]B$.
(ii) *If* $\mathcal{D}$ *derives* $\Gamma \vdash e : A$ *and* $\Gamma' \vdash \sigma : \Gamma$, *then there exists* $\mathcal{D}'$ *deriving* $\Gamma' \vdash [\sigma]e : [\sigma]A$. *Moreover, if* $\overline{\Gamma} = \Gamma$ *(that is,* Γ *contains only index variables and index constraints), there is such a* $\mathcal{D}'$ *not larger than* $\mathcal{D}$ *(that is, the number of typing rules used in* $\mathcal{D}'$ *is at most the number used in* $\mathcal{D}$*).*
(iii) *If* $\Gamma \vdash e : B$ *and* $\Gamma, f{:}B \vdash e' : A$ *then* $\Gamma \vdash [e/f]\, e' : A$.

Similar properties hold for matches ms.

Proof. By induction on the respective derivations. □

6.2 Definiteness

We formalize definiteness as follows, for typing judgments with (possibly) non-empty contexts:

Definition 1. *A typing judgment* $\Gamma \vdash e : A$ *is definite with respect to a substitution* $\vdash \sigma : \Gamma$ *if and only if*

(i) $\vdash [\sigma]A \leq \bot$ *is not derivable;*
(ii) *if* $\vdash [\sigma]A \leq B_1 \vee B_2$ *then* $\vdash [\sigma]e : B_1$ *or* $\vdash [\sigma]e : B_2$*;*
(iii) *if* $\vdash [\sigma]A \leq \Sigma b{:}\gamma.\, B$, *there exists an index* $\vdash i : \gamma$ *where* $\vdash [\sigma]e : [i/b]\, B$.

Definition 2. *A substitution* $\vdash \sigma : \Gamma$ *is definite iff for all* A *such that* $\Gamma(x) = A$, $\Gamma \vdash x : A$ *is definite with respect to* σ.

To prove value definiteness (Theorem 2), which is central in the proof of type safety, we first prove a weaker form of the theorem that depends on the definiteness of the substitution σ. This weak form will allow us to show that *every* well-typed substitution is definite, which will lead directly to a proof of the theorem.

Lemma 4 (Weak value definiteness). *If* $\Gamma \vdash v : A$ *where* σ *is a definite substitution and* $\vdash \sigma : \Gamma$, *then* $\Gamma \vdash v : A$ *is definite with respect to* σ, *that is:*

(i) *it is not the case that* $\Gamma \vdash A \leq \bot$*;*
(ii) *if* $\Gamma \vdash A \leq A_1 \vee A_2$ *then* $\vdash [\sigma]v : [\sigma]A_1$ *or* $\vdash [\sigma]v : [\sigma]A_2$*;*
(iii) *if* $\Gamma \vdash A \leq \Sigma b{:}\gamma.\, B$ *then there exists* $\vdash i : \gamma$ *where* $\vdash [\sigma]v : [\sigma, i/b]\, B$.

Proof. By induction on the derivation of $\Gamma \vdash v : A$. The term v is a value, so we need not consider rules that cannot type values. Furthermore, the (contra) case cannot arise. Most cases follow easily from the IH and properties of subtyping (reflexivity, transitivity, inversion). For (var) we use the fact that σ is a definite substitution. That leaves only the contextual rules, $(\bot E)$, $(\vee E)$, (direct) and (ΣE); we show the first two cases (the last two are similar to $(\vee E)$):

– **Case** ($\bot E$): $\mathcal{D} = \dfrac{\Gamma \vdash e' : \bot}{\Gamma \vdash E[e'] : A}\ (\bot E)$

  $E[e']$ is a value. By Lemma 2, e' is a value. $\Gamma \vdash \bot \leq \bot$, so by the IH (i), this case cannot arise.
– **Case** ($\vee E$):

  $\mathcal{D} = \dfrac{\Gamma \vdash e' : C_1 \vee C_2 \quad \Gamma, x{:}C_1 \vdash E[x] : A \quad \Gamma, y{:}C_2 \vdash E[y] : A}{\Gamma \vdash E[e'] : A}\ (\vee E)$

  $E[e']$ value is given. By Lemma 2, $E[x]$ value and e' value. By the IH, either $\vdash [\sigma]e' : [\sigma]C_1$ or $\vdash [\sigma]e' : [\sigma]C_2$. Assume the first possibility, $\vdash [\sigma]e' : C_1$ (the second is symmetric). Let $\sigma' = \sigma, [\sigma]e'/x$; by (pvar-$\sigma$), $\vdash \sigma' : \Gamma, x{:}C_1$. By the IH, $\Gamma \vdash e' : C_1 \vee C_2$ is definite, so σ' is a definite substitution. $E[x]$ value so we can apply the IH to show that $[\sigma']\,E[x]$ has properties (i), (ii), (iii). But $[\sigma']\,E[x] = [\sigma]\,E[e']$, so $[\sigma]\,E[e']$ has properties (i), (ii), (iii).

□

Lemma 5. *Every substitution σ such that $\vdash \sigma : \Gamma$ is definite.*

Proof. By induction on the derivation $\mathcal{D}$ of $\vdash \sigma : \Gamma$. The case for (empty-σ) is trivial. The cases for (prop-σ) and (ivar-σ) follow easily from the IH. For (pvar-σ) deriving $\vdash \sigma, v/x : \Gamma, x{:}A$, we have $\vdash v : [\sigma]A$ as a subderivation. Since v is closed, $v = [\sigma]v = [\sigma, v/x]x$, yielding $\vdash [\sigma, v/x]x : [\sigma]A$. The result follows by Lemma 4 applied with an empty substitution. □

Theorem 2 (Value definiteness). *If $\vdash \sigma : \Gamma$ and $\Gamma \vdash v : A$, then:*

(i) it is not the case that $\Gamma \vdash A \leq \bot$;
(ii) if $\Gamma \vdash A \leq A_1 \vee A_2$ then $\vdash [\sigma]v : [\sigma]A_1$ or $\vdash [\sigma]v : [\sigma]A_2$;
(iii) if $\Gamma \vdash A \leq \Sigma a{:}\gamma.\, A'$, there exists an index $\vdash i : \gamma$ where $\vdash [\sigma]\, v : [\sigma, i/a]\, A'$.

Proof. Follows immediately from Lemmas 4 and 5. □

For each ordinary type (not property types) we have a value inversion lemma (also known as *genericity* or *canonical forms*). We show only one example. Note the necessary generalization to allow for substitutions.

Lemma 6 (Inversion on $\rightarrow$).
If $\mathcal{D}$ derives $\Gamma \vdash v : B$ and $\Gamma \vdash B \leq B_1 \rightarrow B_2$ and $\vdash \sigma : \Gamma$ then $v = \lambda x.\, e$ where $\vdash [\sigma, v'/x]\, e : [\sigma]B_2$ for any $\vdash v' : [\sigma]B_1$.

Proof. By induction on $\mathcal{D}$. □

7 Type Preservation and Progress

Having proved value definiteness, we are ready to prove type safety. We prove the preservation and progress theorems simultaneously; we could prove them separately, but the proofs would share so much structure as to be more cumbersome than the simultaneous proof. (Our semantics is deterministic, so the combined form is meaningful.)

Theorem 3 (Type Preservation and Progress). *If $\Gamma \vdash e : C$ and σ is a substitution over program variables such that $\vdash \sigma : \Gamma$ and $\overline{\Gamma} = \cdot$, then either*

(1) e value and $\vdash [\sigma]\, e : C$, or
(2) there exists a term e' such that $[\sigma]\, e \mapsto e'$ and $\vdash e' : C$.

(By $\vdash \sigma : \Gamma$, σ substitutes no fixed point variables, so Γ must contain no fixed point variables. Moreover, $\overline{\Gamma} = \cdot$ so Γ contains no index variables.)

Proof. By induction on the derivation $\mathcal{D}$ of $\Gamma \vdash e : C$. Note that since Γ contains no index variables, $[\sigma]A = A$ for all types A. If e value, the result follows by Lemma 3. So suppose e is not a value. Rules ($\mathbf{1}I$), ($\to I$), ($\wedge I$), ($\top I$), (ΠI) and (var) can only type values. Γ types no fixed point variable, so (fixvar) cannot have been used. $\vdash \sigma : \Gamma$, so $\overline{\Gamma} \not\models \bot$: The cases for (sub) and (**fix**) use Lemma 3. The ($\to E$) case requires Lemmas 6 and 3. For ($\vee I_1$), ($\vee I_2$), (ΣI), ($\wedge E_1$), ($\wedge E_2$) simply apply the IH and reapply the rule. For (direct), ($\bot E$), ($\vee E$) and (ΣE), which type an evaluation context $E[e']$, we proceed thus:

- If the whole term $E[e']$ is a value, just apply Lemma 3.
- If e' is not a value:
  (1) apply the IH to $\Gamma \vdash e' : D$ to obtain $[\sigma]\, e' \mapsto e''$ with $\vdash e'' : D$;
  (2) from $[\sigma]\, e' \mapsto e''$, use (ev-context) to show $[\sigma]\, E[e'] \mapsto [\sigma]\, E[e'']$;
  (3) reapply the rule, with premise $\vdash e'' : D$, to yield $\vdash [\sigma]\, E[e''] : C$.
- If e' is a value (but $E[e']$ is not), use value definiteness (Theorem 2), yielding a contradiction for ($\bot E$), or a new derivation for (direct), ($\vee E$), (ΣE); in the latter cases apply the IH with substitution $[\sigma, [\sigma]e'/x]$.

The last subcase is the most interesting; we show it for ($\bot E$) and ($\vee E$). The (direct) and (ΣE) cases are similar.

- **Case ($\bot E$):** $\mathcal{D} = \dfrac{\Gamma \vdash e' : \bot}{\Gamma \vdash E[e'] : C}\ (\bot E)$

  e' value and $\vdash \sigma : \Gamma$ are given. We have $\Gamma \vdash e' : \bot$ as a subderivation. By Theorem 2, $\Gamma \not\vdash e' : \bot$, a contradiction.

- **Case ($\vee E$):** $\mathcal{D} = \dfrac{\Gamma \vdash e' : A \vee B \quad \Gamma, x{:}A \vdash E[x] : C \quad \Gamma, y{:}B \vdash E[y] : C}{\Gamma \vdash E[e'] : C}\ (\vee E)$

  e' value is given. We have $\Gamma \vdash e' : A \vee B$ as a subderivation. By Theorem 2, either $\vdash [\sigma]e' : A$ or $\vdash [\sigma]e' : B$. Since e' value, $[\sigma]e'$ value. Assume $\vdash [\sigma]e' : A$ (the other case is symmetric). $\Gamma, x{:}A \vdash E[x] : C$ is a subderivation. Let $\sigma' = \sigma, [\sigma]e'/x$. It is given that $\vdash \sigma : \Gamma$ and $\vdash [\sigma]\, e' : A$. By (pvar-σ), $\sigma' : \Gamma, x{:}A$, so by the IH, $[\sigma']\, E[x] \mapsto e''$ and $\vdash e'' : C$. But $[\sigma']\, E[x] = [\sigma, [\sigma']e'/x]\, E[x] = [\sigma]\, E[e']$, yielding $[\sigma]\, E[e'] \mapsto e''$.

□

8 Related Work

The notion of datasort refinement combined with intersection types was introduced by Freeman and Pfenning [8]. They showed that full type inference was decidable under the so-called refinement restriction by using techniques from abstract interpretation. Interaction with effects in a call-by-value language was first addressed conclusively by Davies and Pfenning [6] which introduced the value restriction on intersection introduction, pointed out the unsoundness of distributivity, and proposed a practical bidirectional checking algorithm.

A different kind of refinement using indexed and dependent function types with indices drawn from a decidable constraint domain was proposed by Xi and Pfenning [17]. This language did not introduce pure property types, requiring syntactic markers for elimination of the existentials. Since this was unrealistic for many programs, Xi [15] presents an algorithm allowing existential elimination at every binding site after translation to a let-normal form. One can see some of the results in the current paper as a *post hoc* justification for this strategy (see the remarks at the end of Section 4.4).

Intersection types [4] were first incorporated into practical languages by Reynolds [12]. Pierce [11] gave examples of programming with intersection and union types in a pure λ-calculus using a type-checking mechanism that relied on syntactic markers. The first systematic study of unions in a type assignment framework by Barbanera, Dezani-Ciancaglini and de'Liguoro [2] identified a number of problems, including the failure of type preservation even for the pure λ-calculus when the union elimination rule is too unrestricted. It also provided a framework for our more specialized study of a call-by-value language with possible effects. van Bakel et al. [13] showed that the minimal relevant logic $\mathsf{B}+$ yields a type assignment system for the pure call-by-value λ-calculus; conjunction and disjunction become intersection and union, respectively. In their $\vee$-elimination rule, the subexpression may appear multiple times but must be a value; this rule is sound but impractical (see Section 4.1).

Some work on program analysis in compilation uses forms of intersection and union types to infer control flow properties [14,10]. Because of the goals of these systems for program analysis and control flow information, the specific forms of intersection and union types are quite different from the ones considered here. Systems of *soft typing* designed for type inference in dynamically typed languages [3] are somewhat similar and also allow intersection, union, and even conditional types [1]. Again, however, the different setting and goals mean that the technical realization differs substantially from our proposal here.

9 Conclusion

We have designed a system of property types for the purpose of checking program invariants in call-by-value languages. We have presented the system as it was designed: incrementally, with each type constructor added orthogonally to an intermediate system that is itself sound and logically motivated. For both the

definite and indefinite types, we have formulated rules that are not only sound but internally regular: the differences among $(\vee E)$, $(\bot E)$, (ΣE), (direct) are logical consequences of the type constructor's arity. The remarkable feature shared by all four rules is that *typing proceeds in evaluation order*, constituting a less *ad hoc* alternative to Xi's conversion to let-normal form. Lastly, we have formulated properties of *definiteness* of judgments, substitutions, and values, vital for our proof of type safety.

The pure type assignment system presented here is undecidable. We are in the process of developing a decidable bidirectional version (extending the system in [7], which did not include $\vee$ and Σ). The present system can be used to verify progress and preservation after erasure of all type annotations, and will be the basis of soundness in the bidirectional system. In particular, it verifies that type-checked programs do not need to carry types at runtime.

The major items of future work are the development of an efficient algorithm for type-checking and the evaluation of the pragmatics of the system in a full-scale language. While we have elided any explicit effects from the present system for the sake of brevity, the analysis in [6] applies to this setting and the present system. Moreover, since parametric polymorphism was orthogonal in the system of [6], we expect polymorphism will be orthogonal here as well. Ultimately, the system must be justified not only by its soundness and internal design but by its effectiveness in checking interesting properties of real programs.

Acknowledgments. This work is supported in part by the National Science Foundation under grants ITR/SY+SI 0121633 *Language Technology for Trustless Software Dissemination* and CCR-0204248 *Type Refinements*. In addition, the first author is supported in part by an NSF Graduate Research Fellowship. Brigitte Pientka and the anonymous referees provided insightful feedback.

References

1. Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. 21st Symposium on Principles of Programming Languages (POPL '94)*, pages 163–173, 1994.
2. Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: syntax and semantics. *Inf. and Comp.*, 119:202–230, 1995.
3. R. Cartwright and M. Fagan. Soft typing. In *Proc. SIGPLAN '91 Conf. Programming Language Design and Impl. (PLDI)*, volume 26, pages 278–292, 1991.
4. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift f. math. Logik und Grundlagen d. Math.*, 27:45–58, 1981.
5. Rowan Davies. Practical refinement-type checking. PhD thesis proposal, Carnegie Mellon University, 1997.
6. Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Proc. Int'l Conf. Functional Programming (ICFP '00)*, pages 198–208, 2000.
7. Joshua Dunfield. Combining two forms of type refinements. Technical Report CMU-CS-02-182, Carnegie Mellon University, September 2002.

8. Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proc. SIGPLAN '91 Conf. Programming Language Design and Impl. (PLDI)*, volume 26, pages 268–277. ACM Press, June 1991.
9. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
10. Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *J. Functional Programming*, 11(3):263–317, 2001.
11. Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.
12. John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.
13. S. van Bakel, M. Dezani-Ciancaglini, U. de'Liguoro, and Y. Motohoma. The minimal relevant logic and the call-by-value lambda calculus. To appear, July 1999.
14. J.B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *J. Functional Programming*, 12(3):183–317, May 2002.
15. Hongwei Xi. *Dependent types in practical programming*. PhD thesis, Carnegie Mellon University, 1998.
16. Hongwei Xi. Dependently typed data structures. Revised version superseding that of WAAAPL '99, available electronically, February 2000.
17. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. 26th Symp. on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, 1999.

Cones and Foci for Protocol Verification Revisited*

Wan Fokkink[1,2] and Jun Pang[1]

[1] CWI, Department of Software Engineering, PO Box 94079,
1090 GB Amsterdam, The Netherlands,
{wan,pangjun}@cwi.nl
[2] Vrije Universiteit Amsterdam, Department of Theoretical Computer Science,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands,
wanf@cs.vu.nl

Abstract. We define a cones and foci proof method, which rephrases the question whether two system specifications are branching bisimilar in terms of proof obligations on relations between data objects. Compared to the original cones and foci method from Groote and Springintveld [22], our method is more generally applicable, and does not require a preprocessing step to eliminate τ-loops. We prove soundness of our approach and give an application.

1 Introduction

In order to make data a first class citizen in the study of processes, the language μCRL [21] combines the process algebra ACP [3] with equational abstract data types [27]. Processes are intertwined with data: Actions and recursion variables are parametrized by data types; an if-then-else construct allows data objects to influence the course of a process; and alternative quantification sums over possibly infinite data domains. Internal activity of a process can be hidden by a hiding operator τ_I, which renames all internal actions (i.e., the actions in the set I) into the hidden action τ [5].

A labeled transition system is associated to each μCRL specification. Two μCRL specifications are considered equivalent if the initial states of their labeled transition systems are branching bisimilar [16]. Verification of system correctness boils down to checking whether the implementation of a system (with all internal activity hidden) is branching bisimilar to the specification of the desired external behavior of the system. Checking whether two states are branching bisimilar can be performed efficiently [23]. The μCRL toolset [7] supports the generation of labeled transition systems, together with reduction modulo branching bisimulation equivalence, and allows model checking of temporal logic formulas [10] via a back-end to the CADP toolset [12].

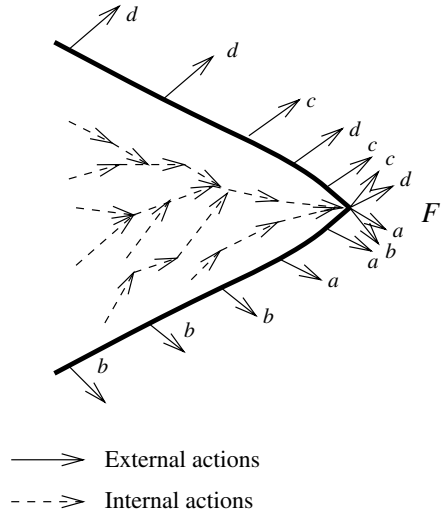
* This research is supported by the Dutch Technology Foundation STW under the project CES5008: Improving the quality of embedded systems using formal design and systematic testing.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 267–281, 2003.

This approach to verify system correctness has three important drawbacks. First, the labeled transition systems of the μCRL specifications involved must be generated; often the labeled transition system of the implementation of a system cannot be generated, as it is too large, or even infinite. Second, this generation usually requires a specific choice for one network or data domain; in other words, only the correctness of an instantiation of the system is proved. Third, support from and rigorous formalization by theorem provers and proof checkers is not readily available.

Groote and Springintveld [22] introduced the *cones and foci* method, which rephrases the question whether two μCRL specifications are branching bisimilar in terms of proof obligations on relations between data objects. These proof obligations can be derived by means of algebraic calculations, in general with the help of invariants (i.e., properties of the reachable states) that are proved separately. This method was used in the verification of a considerable number of real-life protocols (e.g., [15, 20, 34]), often with the support of a theorem prover or proof checker.

The main idea of this method is that quite often in the implementation of a system, internal actions progress inertly towards a state in which no internal actions can be executed; such a state is declared to be a *focus point*. The *cone* of a focus point consists of the states that can reach this focus point by a string of internal actions. In the absence of infinite sequences of internal actions, each state belongs to a cone. This core idea is depicted below. Note that the external actions at the edge of the depicted cone can also be executed in the ultimate focus point F; this is essential for soundness of the cones and foci method, as otherwise internal actions in the cone would not be inert.



Linear process equations [6] constitute a restricted class of μCRL specifications in some kind of linear format. Algorithms have been developed to transform μCRL specifications into this linear format [19, 24, 35]. In a linear process equation, the states of the associated labeled transition system are data objects.

Assume that the implementation of a system and its desired external behavior are both given in the form of a linear process equation. In the cones and foci method, a *state mapping* ϕ relates each state of the implementation to a state of the desired external behavior. Groote and Springintveld [22] formulated *matching criteria*, consisting of relations between data objects, which ensure that states s and $\phi(s)$ are branching bisimilar.

If an implementation, with all internal activity hidden, includes infinite sequences of τ-actions, then Groote and Springintveld [22] distinguish between *progressing* and *non-progressing* τ-actions. Their requirements are that (1) there is no infinite sequence of progressing τ-actions, (2) non-progressing τ-actions are only executed at a focus point, and (3) a focus point cannot perform progressing τ-actions. A *pre-abstraction function* divides occurrences of τ-actions in the implementation into progressing and non-progressing ones, and only progressing τ's are abstracted away; in many cases it is far from trivial to define the proper pre-abstraction. Finally, a special *fair abstraction rule* [2] can be used to try and eliminate the remaining (non-progressing) τ's.

In this paper, we propose an adaptation of the cones and foci method, in which the cumbersome treatment of infinite sequences of τ-actions is no longer necessary. This improvement of the cones and foci method was conceived during the verification of a sliding window protocol [13], where the adaptation simplified matters considerably. As before, the method deals with linear process equations, requires the definition of a state mapping, and generates the same matching criteria. However, we allow the user to freely assign which states are focus points (instead of prescribing that they are the states in which no progressing τ-actions can be performed), as long as each state is in the cone of a focus point. We do allow infinite sequences of internal actions. Since the meaning of recursive specifications that include infinite sequences of τ-actions is ambiguous, we leave the hiding operator τ_I around the μCRL specification of the implementation in place. No distinction between progressing and non-progressing internal actions is needed, and loops of internal actions are eliminated without having to resort to a fair abstraction rule.

We prove that our method is sound modulo branching bisimulation equivalence. Furthermore, we apply our method to the Concurrent Alternating Bit Protocol [26], which served as the main example in [22]. While the old cones and foci method required a typical cumbersome treatment of τ-loops, here we can take these τ-loops in our stride.

Related Work In compiler correctness, advances have been made to validate programs at a symbolic level with respect to an underlying simulation notion (e.g., [9, 17, 30]). The methodology surrounding cones and foci incorporates well-known and useful concepts such as the precondition/effect notation [25, 28], invariants and simulations. Linear process equations resemble the UNITY format [8] and recursive applicative program schemes [11]; state mappings are comparable to refinement mappings [29, 32] and simulation [14]. Van der Zwaag [36] gave an adaptation of the cones and foci method from [22] to a timed setting, modulo timed branching bisimulation equivalence. We leave it as an

open question whether our innovations for the cones and foci method can also be introduced in this timed setting.

2 Preliminaries

2.1 μCRL

μCRL [21] is a language for specifying distributed systems and protocols in an algebraic style. It is based on process algebra extended with equational abstract data types. In a μCRL specification, one part specifies the data types, while a second part specifies the process behavior. We do not describe the treatment of data types in μCRL in detail. For our purpose it is sufficient that processes can be parametrized with data. We assume the data sort of booleans *Bool* with constant T and F, and the usual connectives $\wedge$, $\vee$, $\neg$ and $\Rightarrow$. For a boolean b, we abbreviate $b = \mathsf{T}$ to b and $b = \mathsf{F}$ to $\neg b$.

The specification of a process is constructed from action names, recursion variables and process algebraic operators. Actions and recursion variables carry zero or more data parameters. There are two predefined actions in μCRL: δ represents deadlock, and τ a hidden action. These two actions never carry data parameters.

Processes are represented by process terms, which describe the order in which the actions from a set *Act* may happen. A process term consists of action names and recursion variables combined by process algebraic operators. $p{\cdot}q$ denotes sequential composition and $p+q$ non-deterministic choice, summation $\sum_{d:D} p(d)$ provides the possibly infinite choice over a data type D, and the conditional construct $p \lhd b \rhd q$ with b a data term of sort *Bool* behaves as p if b and as q if $\neg b$. Parallel composition $p \parallel q$ interleaves the actions of p and q; moreover, actions from p and q may also synchronize to a communication action, when this is explicitly allowed by a predefined communication function. Two actions can only synchronize if their data parameters are semantically the same, which means that communication can be used to represent data transfer from one system component to another. Encapsulation $\partial_H(p)$, which renames all occurrences in p of actions from the set H into δ, can be used to force actions into communication. Finally, hiding $\tau_I(p)$ renames all occurrences in p of actions from the set I into τ. The syntax and semantics of μCRL are given in [21].

2.2 Labeled Transition Systems

Labeled transition systems (LTSs) capture the operational behavior of concurrent systems. An LTS consists of transitions $s \xrightarrow{a} s'$, denoting that the state s can evolve into the state s' by the execution of action a. To each μCRL specification belongs an LTS, defined by the structural operational semantics for μCRL in [21].

Definition 1 (Labeled transition system). *A* labeled transition system *is a tuple* $(S, Lab, \rightarrow, s_0)$*, where* S *is a set of states,* Lab *a set of transition labels,*

$\rightarrow \subseteq S \times Lab \times S$ *a transition relation, and* s_0 *the initial state. A transition* (s, ℓ, s') *is denoted by* $s \xrightarrow{\ell} s'$.

Here, S consists of μCRL specifications, and Lab consists of actions from $Act \cup \{\tau\}$ parametrized by data. We define *branching bisimilarity* [16] between states in LTSs. Branching bisimulation is an equivalence relation [4].

Definition 2 (Branching bisimulation). *Assume an LTS. A symmetric binary relation* B *on states is a* branching bisimulation *if* sBt *and* $s \xrightarrow{\ell} s'$ *implies:*

- *either* $\ell = \tau$ *and* $s' \mathcal{B}\, t$;
- *or there is a sequence of (zero or more)* τ*-transitions* $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_0$ *such that* sBt_0 *and* $t_0 \xrightarrow{\ell} t'$ *with* $s'Bt'$.

Two states s *and* t *are* branching bisimilar*, denoted by* $s \underline{\leftrightarrow}_b t$*, if there is a branching bisimulation relation* $\mathcal{B}$ *such that* $s \mathcal{B}\, t$.

The μCRL toolset [7] supports the generation of labeled transition systems of μCRL specifications, together with reduction modulo branching bisimulation equivalence and model checking of temporal logic formulas. This approach has been used to analyze a wide range of protocols and distributed systems (e.g., [1, 18, 31, 33]).

In this paper we focus on analyzing protocols and distributed systems on the level of their symbolic specifications.

2.3 Linear Process Equations

A *linear process equation* (LPE) is a one-line μCRL specification consisting of actions, summations, sequential compositions and conditional constructs. In particular, an LPE does not contain any parallel operators, encapsulations or hidings. In essence an LPE is a vector of data parameters together with a list of condition, action and effect triples, describing when an action may happen and what is its effect on the vector of data parameters. Each μCRL specification that does not include successful termination can be transformed into an LPE [35].[1]

Definition 3 (Linear process equation). *A* linear process equation *is a* μ*CRL specification of the form*

$$X(d{:}D) = \sum_{a \in Act \cup \{\tau\}} \sum_{e:E} a(f_a(d,e)) \cdot X(g_a(d,e)) \lhd h_a(d,e) \rhd \delta$$

where $f_a : D \times E \rightarrow D$, $g_a : D \times E \rightarrow D$ *and* $h_a : D \times E \rightarrow Bool$ *for each* $a \in Act \cup \{\tau\}$.

[1] To cover μCRL specifications with successful termination, LPEs should also include a summand $\sum_{a \in Act \cup \{\tau\}} \sum_{e:E} a(f_a(d,e)) \lhd h_a(d,e) \rhd \delta$. The cones and foci method extends to this setting without any complication. However, this extension would complicate the matching criteria in Definition 7. For the sake of presentation, successful termination is not taken into account here.

The LPE in Definition 3 has exactly one LTS as its solution.[2] In this LTS, the states are data elements $d{:}D$ (where D may be a Cartesian product of n data types, meaning that d is a tuple $(d_1, ..., d_n)$) and the transition labels are actions parametrized with data. The LPE expresses that state d can perform $a(f_a(d, e))$ to end up in state $g_a(d, e)$, under the condition that $h_a(d, e)$ is true. The data type E gives LPEs a more general form, as not only the data parameter $d{:}D$ but also the data parameter $e{:}E$ can influence the parameter of action a, the condition h_a and the resulting state g_a.

Definition 4 (Invariant). *A mapping $\mathcal{I} : D \to Bool$ is an invariant for an LPE, written as in Definition 3, if for all $a \in Act \cup \{\tau\}$, $d{:}D$ and $e{:}E$,*

$$\mathcal{I}(d) \wedge h_a(d, e) \;\Rightarrow\; \mathcal{I}(g_a(d, e)).$$

Intuitively, an invariant characterizes the set of reachable states of an LPE. That is, if $\mathcal{I}(d)$, and if one can involve from state d to state d' in zero or more transitions, then $\mathcal{I}(d')$. Namely, if $\mathcal{I}$ holds in state d and it is possible to execute $a(f_a(d, e))$ in this state (meaning that $h_a(d, e)$), then it is ensured that $\mathcal{I}$ holds in the resulting state $g_a(d, e)$. Invariants tend to play a crucial role in algebraic verifications of system correctness that involve data.

3 Cones and Foci

In this section, we present our version of the cones and foci method [22]. Suppose that we have an LPE $X(d{:}D)$ (including internal actions from a set I, which will be hidden) specifying the implementation of a system, and an LPE $Y(d'{:}D')$ (without internal actions) specifying the desired input/output behavior of this system. Furthermore, assume an invariant $\mathcal{I} : D \to Bool$ characterizing the reachable states of X. We want to prove that the implementation exhibits the desired input/output behavior.

We assume the presence of an invariant $\mathcal{I} : D \to Bool$ for X. In the cones and foci method, a *state mapping* $\phi : D \to D'$ relates each state of the implementation X to a state of the desired external behavior Y. Furthermore, some states in D are designated to be *focus points*. In contrast with the approach of [22], we allow to freely assign focus points, as long as each state $d{:}D$ of X with $\mathcal{I}(d)$ can reach a focus point by a sequence of internal transitions. If a number of *matching criteria* for $d{:}D$ are fulfilled, consisting of relations between data objects, and if $\mathcal{I}(d)$, then the states d and $\phi(d)$ are guaranteed to be branching bisimilar. These matching criteria require that (A) after hiding, all internal transitions of d become invisible, (B) each external transition of d can be mimicked by $\phi(d)$, and (C) if d is a focus point, then vice versa each transition of $\phi(d)$ can be mimicked by d.

We start with defining the predicate *FC*, designating the focus points of X in D. Next we define the state mapping together with its matching criteria.

[2] LPEs exclude "unguarded" recursive specifications such as $X = X$, which have multiple solutions.

Definition 5 (Focus point). *A* focus condition *is a mapping* $FC : D \to Bool$. *If* $FC(d)$, *then* d *is called a* focus point.

Definition 6 (State mapping). *A* state mapping *is of the form* $\phi : D \to D'$.

Definition 7 (Matching criteria). *Let the LPE* X *be of the form*

$$X(d{:}D) = \sum_{a \in Act} \sum_{e:E} a(f_a(d,e)) \cdot X(g_a(d,e)) \lhd h_a(d,e) \rhd \delta.$$

Furthermore, let the LPE Y *be of the form*

$$Y(d'{:}D') = \sum_{a \in Act \setminus I} \sum_{e:E} a(f'_a(d',e)) \cdot Y(g'_a(d',e)) \lhd h'_a(d',e) \rhd \delta.$$

A state mapping $\phi : D \to D'$ *satisfies the* matching criteria *for* $d{:}D$ *if for all* $a \in Act \setminus I$ *and* $c \in I$*:*

$$\begin{array}{ll} I & \forall e{:}E\,(h_c(d,e) \Rightarrow \phi(d) = \phi(g_c(d,e)));\\ II & \forall e{:}E\,(h_a(d,e) \Rightarrow h'_a(\phi(d),e));\\ III & FC(d) \Rightarrow \forall e{:}E\,(h'_a(\phi(d),e) \Rightarrow h_a(d,e));\\ IV & \forall e{:}E\,(h_a(d,e) \Rightarrow f_a(d,e) = f'_a(\phi(d),e));\\ V & \forall e{:}E\,(h_a(d,e) \Rightarrow \phi(g_a(d,e)) = g'_a(\phi(d),e)). \end{array}$$

Matching criterion I requires that after hiding, all internal c-transitions from d are invisible, meaning that d and $g_c(d,e)$ are branching bisimilar. Criteria II, IV and V express that each external transition of d can be simulated by $\phi(d)$. Finally, criterion III expresses that if d is a focus point, then each external transition of $\phi(d)$ can be simulated by d.

Theorem 1. *Assume LPEs* $X(d{:}D)$ *and* $Y(d'{:}D')$ *written as in Definition 7. Let* $I \subseteq Act$, *and let* $\mathcal{I} : D \to Bool$ *be an invariant for* X. *Suppose that for all* $d{:}D$ *with* $\mathcal{I}(d)$,

1. $\phi : D \to D'$ *satisfies the matching criteria for* d, *and*
2. *there is a* $\hat{d}{:}D$ *such that* $FC(\hat{d})$ *and* $d \xrightarrow{c_1} \cdots \xrightarrow{c_n} \hat{d}$ *with* $c_1, \ldots, c_n \in I$ *in the LTS for* X.

Then for all $d{:}D$ *with* $\mathcal{I}(d)$,

$$\tau_I(X(d)) \leftrightarrows_b Y(\phi(d)).$$

Proof. We assume without loss of generality that D and D' are disjoint. Define $B \subseteq D \cup D' \times D \cup D'$ as the smallest relation such that whenever $\mathcal{I}(d)$ for a $d{:}D$ then $dB\phi(d)$ and $\phi(d)Bd$. Clearly, B is symmetric. We show that B is a branching bisimulation relation.

Let sBt and $s \xrightarrow{\ell} s'$. First consider that case where $\phi(s) = t$. By definition of B we have $\mathcal{I}(s)$.

1. If $\ell = \tau$, then $h_c(s,e)$ and $s' = g_c(s,e)$ for some $c \in I$ and $e{:}E$. By matching criterion I, $\phi(g_c(s,e)) = t$. Moreover, $\mathcal{I}(s)$ and $h_c(s,e)$ together imply $\mathcal{I}(g_c(s,e))$. Hence, $g_c(s,e)Bt$.
2. If $\ell \neq \tau$, then $h_a(s,e)$, $s' = g_a(s,e)$ and $\ell = a(f_a(s,e))$ for some $a \in Act\backslash I$ and $e{:}E$. By matching criteria II and IV, $h'_a(t,e)$ and $f_a(s,e) = f'_a(t,e)$. Hence, $t \stackrel{a(f_a(s,e))}{\rightarrow} g'_a(t,e)$. Moreover, $\mathcal{I}(s)$ and $h_a(s,e)$ together imply $\mathcal{I}(g_a(s,e))$, and matching criterion V yields $\phi(g_a(s,e)) = g'_a(t,e)$, so $g_a(s,e)Bg'_a(t,e)$.

Next consider the case where $s = \phi(t)$. Since $s \stackrel{\ell}{\rightarrow} s'$, for some $a \in Act\backslash I$ and $e{:}E$, $h'_a(s,e)$, $s' = g'_a(s,e)$ and $\ell = a(f'_a(s,e))$. By definition of B we have $\mathcal{I}(t)$.

1. If $FC(t)$, then by matching criterion III, $h_a(t,e)$. So by matching criterion IV, $f_a(t,e) = f'_a(s,e)$. Hence, $t \stackrel{a(f'_a(s,e))}{\rightarrow} g_a(t,e)$. Moreover, $\mathcal{I}(t)$ and $h_a(t,e)$ together imply $\mathcal{I}(g_a(t,e))$, and matching criterion V yields $\phi(g_a(t,e)) = g'_a(s,e)$, so $g'_a(s,e)Bg_a(t,e)$.
2. If $\neg FC(t)$, then there is a $\hat{t}{:}D$ with $FC(\hat{t})$ such that $t \stackrel{c_1}{\rightarrow} ... \stackrel{c_n}{\rightarrow} \hat{t}$ with $c_1, \ldots, c_n \in I$ in the LTS for X. This implies that $t \stackrel{\tau}{\rightarrow} ... \stackrel{\tau}{\rightarrow} \hat{t}$ in the LTS for $\tau_I(X)$. Invariant $\mathcal{I}$, so also the matching criteria, hold for all states on this τ-path. Repeatedly applying matching criterion I we get $\phi(\hat{t}) = \phi(t) = s$. So matching criterion III together with $h'_a(s,e)$ yields $h_a(\hat{t},e)$. Then by matching criterion IV, $f_a(\hat{t},e) = f'_a(s,e)$, so $t \stackrel{\tau}{\rightarrow} ... \stackrel{\tau}{\rightarrow} \hat{t} \stackrel{a(f'_a(s,e))}{\rightarrow} g_a(\hat{t},e)$. Moreover, $\mathcal{I}(\hat{t})$ and $h_a(\hat{t},e)$ together imply $\mathcal{I}(g_a(\hat{t},e))$, and matching criterion V yields $\phi(g_a(\hat{t},e)) = g'_a(s,e)$, so $sB\hat{t}$ and $g'_a(s,e)Bg_a(\hat{t},e)$.

Concluding, B is a branching bisimulation.

We note that Groote and Springintveld [22] proved for their notion of their cones and foci method that it can be derived from the axioms of μCRL, which implies that their method is sound modulo branching bisimulation equivalence. We leave it as future work to try and derive our cones and foci method from the axioms of μCRL.

4 Application to the CABP

Groote and Springintveld [22] proved correctness of the Concurrent Alternating Bit Protocol (CABP) [26] as an application of their cones and foci method. Here we redo their correctness proof using our version of the cones and foci method, where in contrast to [22] we can take loops of internal activity in our stride.

In the CABP, data elements $d_1, d_2, \ldots$ are communicated from a data transmitter S to a data receiver R via a lossy channel, so that a message can be corrupted or lost. Therefore, acknowledgments are sent from R to S again via a lossy channel. In the CABP, sending and receiving of acknowledgments is decoupled from R and S, in the form of separate components AS and AR, respectively,

where AS autonomously sends acknowledgments to AR. This ensures a better use of available bandwidth.

S attaches a bit 0 to data elements d_{2k-1} and a bit 1 to data elements d_{2k}, and AS sends back the attached bit to acknowledge reception. S keeps on sending a pair (d_i, b) until AR receives the bit b and sends the message ac to S; then S starts sending the next pair $(d_{i+1}, 1-b)$. Alternation of the attached bit enables R to determine whether a received datum is really new, and alternation of the acknowledging bit enables AR to determine which datum is being acknowledged.

The CABP contains unbounded internal behavior, which occurs when a channel eternally corrupts or loses the same datum or acknowledgment. The fair abstraction paradigm [2], which underlies branching bisimulation, says that such infinite sequences of faulty behavior do not exist in reality, because the chance of a channel failing infinitely often is zero. Groote and Springintveld [22] defined a pre-abstraction function to hide all internal actions except those that are executed in focus points, and used Koomen's fair abstraction rule [2] to eliminate the remaining loops of internal actions. In our adaptation of the cones and foci method, focus points can perform internal actions, so neither pre-abstraction nor Koomen's fair abstraction rule are needed here.

The structure of the CABP is shown in Figure 1. The CABP system is built from six components.

- S is a *data transmitter*, which reads data from port 1 and transmits such a datum repeatedly via channel K, until an acknowledgment ac regarding this datum is received from AR.
- K is a lossy *data transmission channel*, which transfers data from S to R. Either it delivers the datum correctly, or it can make two sorts of mistakes: lose the datum or change it into the checksum error ce. The non-deterministic choice between the three options is modeled by the action j.
- R is a *data receiver*, which receives data from K, sends freshly received data into port 2, and sends an acknowledgment to AS via port 5.
- AS is an *acknowledgment transmitter*, which receives an acknowledgment from R and repeatedly transmits it via L to AR.
- L is a lossy *acknowledgment transmission channel*, which transfers acknowledgments from AS to AR. Either it delivers the acknowledgment correctly, or it can make two sorts of mistakes: lose the acknowledgment or change it into ae.
- AR is an *acknowledgment receiver*, which receives acknowledgments from L and passes them on to S.

The components can perform read $r_n(...)$ and send $s_n(...)$ actions to transport data through port n. A read and a send action over the same port n can synchronize into a communication action $c_n(...)$. For a detailed description of the data types and each component's specification in μCRL the reader is referred to [22]. The μCRL specification of the CABP is obtained by putting the six components in parallel, encapsulating the internal send and read actions at ports $\{3, 4, 5, 6, 7, 8\}$, and hiding the internal communication actions at these ports together with the action j.

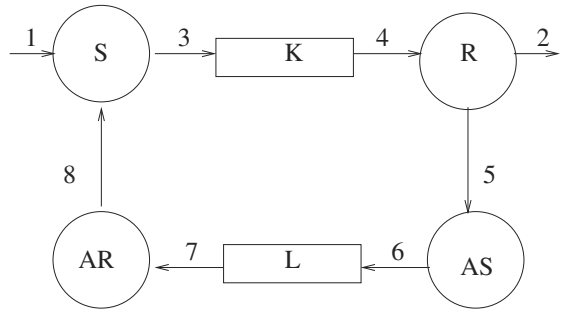


Fig. 1. The structure of the CABP

4.1 Implementation and External Behavior

As a starting point we take the LPE *Sys* that is obtained from the implementation of the CABP; see [22]. It includes the sort *Bit* with elements b_0 and b_1 and with an inversion function $inv : Bit \to Bit$, and the sort *Nat* of natural numbers. The sort D contains the data elements to be transferred by the protocol. $eq : S \times S \to Bool$ coincides with the equality relation between elements of the sort S.

Definition 8. *In each summand of the LPE for Sys below, we only present the parameters whose values are changed.*

$$
\begin{aligned}
&Sys(d_s{:}D, b_s{:}Bit, i_s{:}Nat, i'_s{:}Nat, d_r{:}D, b_r{:}Bit,\\
&\quad i_r{:}Nat, d_k{:}D, b_k{:}Bit, i_k{:}Nat, b_l{:}Bit, i_l{:}Nat)\\
&= \textstyle\sum_{d:D} r_1(d)\cdot Sys(d/d_s, 2/i_s) \lhd eq(i_s,1) \rhd \delta && (1)\\
&+ c_3(\langle d_s, b_s\rangle)\cdot Sys(d_s/d_k, b_s/b_k, 2/i_k) \lhd eq(i_s,2) \wedge eq(i_k,1) \rhd \delta && (2)\\
&+ (j\cdot Sys(1/i_k) + j\cdot Sys(3/i_k) + j\cdot Sys(4/i_k)) \lhd eq(i_k,2) \rhd \delta && (3)\\
&+ c_4(\langle d_k, b_r\rangle)\cdot Sys(d_k/d_r, 2/i_r, 1/i_k) \lhd eq(i_r,1) \wedge eq(b_r,b_k) \wedge eq(i_k,3) \rhd \delta && (4)\\
&+ c_4(\langle d_k, b_r\rangle)\cdot Sys(1/i_k) \lhd eq(i_r,1) \wedge eq(b_r, inv(b_k)) \wedge eq(i_k,3) \rhd \delta && (5)\\
&+ c_4(ce)\cdot Sys(1/i_k) \lhd eq(i_r,1) \wedge eq(i_k,4) \rhd \delta && (6)\\
&+ s_2(d_r)\cdot Sys(3/i_r) \lhd eq(i_r,2) \rhd \delta && (7)\\
&+ c_5(ac)\cdot Sys(inv(b_r)/b_r, 1/i_r) \lhd eq(i_r,3) \rhd \delta && (8)\\
&+ c_6(inv(b_r))\cdot Sys(inv(b_r)/b_l, 2/i_l) \lhd eq(i_l,1) \rhd \delta && (9)\\
&+ (j\cdot Sys(1/i_l) + j\cdot Sys(3/i_l) + j\cdot Sys(4/i_l)) \lhd eq(i_l,2) \rhd \delta && (10)\\
&+ c_7(b_l)\cdot Sys(1/i_l, 2/i'_s) \lhd eq(i'_s,1) \wedge eq(b_l,b_s) \wedge eq(i_l,3) \rhd \delta && (11)\\
&+ c_7(b_l)\cdot Sys(1/i_l) \lhd eq(i'_s,1) \wedge eq(b_l, inv(b_s)) \wedge eq(i_l,3) \rhd \delta && (12)\\
&+ c_7(ae)\cdot Sys(1/i_l) \lhd eq(i'_s,1) \wedge eq(i_l,4) \rhd \delta && (13)\\
&+ c_8(ac)\cdot Sys(inv(b_s)/b_s, 1/i_s, 1/i'_s) \lhd eq(i_s,2) \wedge eq(i'_s,2) \rhd \delta && (14)
\end{aligned}
$$

In the LPE *Sys*, d_s, d_r and d_k represent a datum of sort D which S, R and K read at ports 1, 4 and 3, respectively; b_s, b_r, b_k and b_l are the attached alternating bit for R, S, K and L, respectively. i_s, i'_s, i_r, i_k and i_l are auxiliary parameters that are introduced by the linearization algorithm which transforms the concurrent

specification of the CADP into the LPE *Sys*; these parameters model different states of S, AR, R, K and L, respectively.

The specification of the external behavior of the CABP is a one-datum buffer, which reads a datum at port 1, and sends out this same datum at port 2.

Definition 9. *The LPE of the external behavior of the CABP is*

$$B(d{:}D, b{:}Bool) = \textstyle\sum_{d':D} r_1(d')\cdot B(d', \mathsf{F}) \lhd b \rhd \delta + s_2(d)\cdot B(d, \mathsf{T}) \lhd \neg b \rhd \delta.$$

4.2 Verification

Let Ξ abbreviate $D \times Bit \times Nat \times Nat \times D \times Bit \times Nat \times D \times Bit \times Nat \times Bit \times Nat$. Furthermore, let $\xi{:}\Xi$ denote $(d_s, b_s, i_s, i'_s, d_r, b_r, i_r, d_k, b_k, i_k, b_l, i_l)$.

Invariants $\mathcal{I}_1$-$\mathcal{I}_5$, $\mathcal{I}_7$ below were taken from [22]. $\mathcal{I}_1$-$\mathcal{I}_5$ describe the range of the data parameters i_s, i'_s, i_k, i_r, and i_l, respectively. $\mathcal{I}_6$ says that b_s and b_k remain equal until S gets an acknowledgment from AR. $\mathcal{I}_7$ expresses that each component in Figure 1 either has received information about the datum being transmitted which it must forward, or did not yet receive this information.

Definition 10.

$$\begin{aligned}
&\mathcal{I}_1(\xi) \equiv eq(i_s, 1) \vee eq(i_s, 2)\\
&\mathcal{I}_2(\xi) \equiv eq(i'_s, 1) \vee eq(i'_s, 2)\\
&\mathcal{I}_3(\xi) \equiv eq(i_k, 1) \vee eq(i_k, 2) \vee eq(i_k, 3) \vee eq(i_k, 4)\\
&\mathcal{I}_4(\xi) \equiv eq(i_r, 1) \vee eq(i_r, 2) \vee eq(i_r, 3)\\
&\mathcal{I}_5(\xi) \equiv eq(i_l, 1) \vee eq(i_l, 2) \vee eq(i_l, 3) \vee eq(i_l, 4)\\
&\mathcal{I}_6(\xi) \equiv \neg eq(i_k, 1) \wedge eq(i_s, 2) \Rightarrow eq(b_s, b_k)\\
&\mathcal{I}_7(\xi) \equiv (eq(i_s, 1) \Rightarrow eq(b_s, inv(b_k)) \wedge eq(b_s, b_r) \wedge eq(d_s, d_k)\\
&\qquad\qquad \wedge eq(d_s, d_r) \wedge eq(i'_s, 1) \wedge eq(i_r, 1) \wedge eq(i_k, 1))\\
&\qquad \wedge (eq(b_s, b_k) \Rightarrow eq(d_s, d_k))\\
&\qquad \wedge (eq(i_r, 2) \vee eq(i_r, 3) \Rightarrow eq(d_s, d_r) \wedge eq(b_s, b_r) \wedge eq(b_s, b_k))\\
&\qquad \wedge (eq(b_s, inv(b_r)) \Rightarrow eq(d_s, d_r) \wedge eq(b_s, b_k))\\
&\qquad \wedge (eq(b_s, b_l) \Rightarrow eq(b_s, inv(b_r)))\\
&\qquad \wedge (eq(i'_s, 2) \Rightarrow eq(b_s, b_l)).
\end{aligned}$$

Lemma 1. *$\mathcal{I}_1$, $\mathcal{I}_2$, $\mathcal{I}_3$, $\mathcal{I}_4$, $\mathcal{I}_5$, $\mathcal{I}_6$ and $\mathcal{I}_4 \wedge \mathcal{I}_7$ are invariants of Sys.*

The focus condition for *Sys* is obtained by taking the disjunction of the summands in the LPE in Definition 8 that deal with an external action; these summands are (1) and (7).

Definition 11. *The focus condition for Sys is*

$$FC(\xi) = eq(i_s, 1) \vee eq(i_r, 2).$$

We proceed to prove that each state satisfying the invariants above can reach a focus point by a sequence of internal transitions, carrying labels from $I = \{c_3, c_4, c_5, c_6, c_7, c_8, j\}$.

Lemma 2. *For each $\xi{:}\Xi$ with $\mathcal{I}_n(\xi)$ for $n = 1$-6, there is a $\hat{\xi}{:}\Xi$ such that $FC(\hat{\xi})$ and $\xi \xrightarrow{c_1} \cdots \xrightarrow{c_n} \hat{\xi}$ with $c_1, \ldots, c_n \in I$ in Sys.*

Proof. Let $\neg FC(\xi)$; in view of $\mathcal{I}_1$ and $\mathcal{I}_4$ this implies $eq(i_s, 2) \wedge (eq(i_r, 1) \vee eq(i_r, 3))$. In case $eq(i_r, 3)$, we can perform $c_5(ac)$ at summand (8) to arrive a state with $eq(i_s, 2) \wedge eq(i_r, 1)$. By $\mathcal{I}_3$ and summands (2), (3) and (6), we can perform internal actions to reach a state where $eq(i_s, 2) \wedge eq(i_r, 1) \wedge eq(i_k, 3)$. We distinguish two cases.

1. $eq(b_r, b_k)$.
   We can perform $c_4(\langle d_k, b_r \rangle)$ at summand (4) to reach a focus point.
2. $eq(b_r, inv(b_k))$.
   If $i'_s = 2$, then we can perform $c(ac)$ at summand (14) to reach a focus point, so by $\mathcal{I}_2$ we can assume that $i'_s = 1$. If $eq(i_l, 3) \wedge eq(b_l, b_s)$, then by performing $c_7(b_l)$ at summand (11) followed by $c_8(ac)$ at summand (14) we can reach a focus point. Otherwise, by $\mathcal{I}_5$ and summands (10), (12) and (13) we can reach a state where $eq(i_s, 2) \wedge eq(i'_s, 1) \wedge eq(i_r, 1) \wedge eq(i_k, 3) \wedge eq(i_l, 1)$. We can perform $c_6(inv(b_r))$ at summand (9) followed by j at summand (10) to reach a state where $eq(i_s, 2) \wedge eq(i'_s, 1) \wedge eq(i_r, 1) \wedge eq(i_k, 3) \wedge eq(i_l, 3) \wedge eq(b_l, inv(b_r))$. By $eq(b_r, inv(b_k))$, $\mathcal{I}_6$ and $eq(b_l, inv(b_r))$ we have $eq(b_l, b_s)$. Hence, we can perform $c_7(b_l)$ at summand (11) followed by $c_8(ac)$ at summand (14) to reach a focus point.

The state mapping $\phi : \Xi \to D \times Bool$ is defined by

$$\phi(\xi) = \langle d_s, eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r) \rangle.$$

Note that ϕ is independent of $i'_s, d_r, d_k, b_k, i_k, b_l, i_l$; we write $\phi(d_s, b_s, i_s, b_r, i_r)$.

Theorem 2. *For all $d{:}D$ and $b_0, b_1{:}Bit$,*

$$\tau_I(Sys(d, b_0, 1, 1, d, b_0, 1, d, b_1, 1, b_1, 1)) \leftrightarrows_b B(d, \mathsf{T}).$$

Proof. It is easy to check that $\wedge_{n=1}^{7} \mathcal{I}_n(d, b_0, 1, 1, d, b_0, 1, d, b_1, 1, b_1, 1)$.

We obtain the following matching criteria. For class I, we only need to check the summands (4), (8) and (14), as the other nine summands that involve an initial action leave the values of the parameters in $\phi(d_s, b_s, i_s, b_r, i_r)$ unchanged.

1. $eq(i_r, 1) \wedge eq(b_r, b_k) \wedge eq(i_k, 3) \;\Rightarrow\; \phi(d_s, b_s, i_s, b_r, i_r) = \phi(d_s, b_s, i_s, b_r, 2/i_r)$
2. $eq(i_r, 3) \;\Rightarrow\; \phi(d_s, b_s, i_s, b_r, i_r) = \phi(d_s, b_s, i_s, inv(b_r)/b_r, 1/i_r)$
3. $eq(i_s, 2) \wedge eq(i'_s, 2) \;\Rightarrow\; \phi(d_s, b_s, i_s, b_r, i_r) = \phi(d_s, inv(b_s)/b_s, 1/i_s, b_r, i_r)$

The matching criteria for the other four classes are produced by summands (1) and (7). For class II we get:

1. $eq(i_s, 1) \;\Rightarrow\; eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r)$
2. $eq(i_r, 2) \;\Rightarrow\; \neg(eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r))$

For class III we get:

1. $(eq(i_s, 1) \vee eq(i_r, 2)) \wedge (eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r)) \;\Rightarrow\; eq(i_s, 1)$
2. $(eq(i_s, 1) \vee eq(i_r, 2)) \wedge \neg(eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r)) \;\Rightarrow\; eq(i_r, 2)$

For class IV we get:

1. $\forall d{:}D\,(eq(i_s,1) \;\Rightarrow\; d = d)$
2. $eq(i_r,2) \;\Rightarrow\; d_r = d_s$

Finally, for class V we get:

1. $\forall d{:}D\,(eq(i_s,1) \;\Rightarrow\; \phi(d/d_s, b_s, 2/i_s, b_r, i_r) = \langle d, \mathsf{F}\rangle)$
2. $eq(i_r,2) \;\Rightarrow\; \phi(d_s, b_s, i_s, b_r, 3/i_r) = \langle d_s, \mathsf{T}\rangle$

We proceed to prove the matching criteria.

I.1 Let $eq(i_r,1)$. Then

$$\begin{aligned}\phi(d_s,b_s,i_s,b_r,i_r) &= \langle d_s, eq(i_s,1) \vee eq(1,3) \vee \neg eq(b_s,b_r)\rangle\\ &= \langle d_s, eq(i_s,1) \vee eq(2,3) \vee \neg eq(b_s,b_r)\rangle\\ &= \phi(d_s,b_s,i_s,b_r,2/i_r).\end{aligned}$$

I.2 Let $eq(i_r,3)$. Then by $\mathcal{I}_7$, $eq(b_s,b_r)$. Hence,

$$\begin{aligned}\phi(d_s,b_s,i_s,b_r,i_r) &= \langle d_s, eq(i_s,1) \vee eq(3,3) \vee \neg eq(b_s,b_r)\rangle\\ &= \langle d_s, \mathsf{T}\rangle\\ &= \langle d_s, eq(i_s,1) \vee eq(i_r,3) \vee \neg eq(b_s, inv(b_r))\rangle\\ &= \phi(d_s,b_s,i_s,inv(b_r)/b_r,1/i_r).\end{aligned}$$

I.3 Let $eq(i'_s,2)$. By $\mathcal{I}_7$, $eq(b_s,b_l)$, which together with $\mathcal{I}_7$ yields $eq(b_s,inv(b_r))$. Hence,

$$\begin{aligned}\phi(d_s,b_s,i_s,b_r,i_r) &= \langle d_s, eq(i_s,1) \vee eq(i_r,3) \vee \neg eq(b_s,b_r)\rangle\\ &= \langle d_s, \mathsf{T}\rangle\\ &= \langle d_s, eq(1,1) \vee eq(i_r,3) \vee \neg eq(inv(b_s),b_r)\rangle\\ &= \phi(d_s,inv(b_s)/b_s,1/i_s,b_r,i_r).\end{aligned}$$

II.1 Trivial.

II.2 Let $eq(i_r,2)$. Then clearly $\neg eq(i_r,3)$, and by $\mathcal{I}_7$, $eq(b_s,b_r)$. Furthermore, according to $\mathcal{I}_7$, $eq(i_s,1) \Rightarrow eq(i_r,1)$, so $eq(i_r,2)$ also implies $\neg eq(i_s,1)$.

III.1 If $\neg eq(i_r,2)$, then $eq(i_s,1) \vee eq(i_r,2)$ implies $eq(i_s,1)$. If $eq(i_r,2)$, then by $\mathcal{I}_7$, $eq(b_s,b_r)$, so that $eq(i_s,1) \vee eq(i_r,3) \vee \neg eq(b_s,b_r)$ implies $eq(i_s,1)$.

III.2 If $\neg eq(i_s,1)$, then $eq(i_s,1) \vee eq(i_r,2)$ implies $eq(i_r,2)$. If $eq(i_s,1)$, then the formula $\neg(eq(i_s,1) \vee eq(i_r,3) \vee \neg eq(b_s,b_r))$ is false, so that it implies $eq(i_r,2)$.

IV.1 Trivial.

IV.2 Let $eq(i_r,2)$. Then by $\mathcal{I}_7$, $eq(d_r,d_s)$.

V.1 Let $eq(i_s,1)$. Then by $\mathcal{I}_7$, $eq(i_r,1)$ and $eq(b_s,b_r)$. So for any $d{:}D$,

$$\begin{aligned}\phi(d/d_s,b_s,2/i_s,b_r,i_r) &= \langle d, eq(2,1) \vee eq(1,3) \vee \neg eq(b_s,b_r)\rangle\\ &= \langle d, \mathsf{F}\rangle.\end{aligned}$$

V.2

$$\begin{aligned}\phi(d_s,b_s,i_s,b_r,3/i_r) &= \langle d_s, eq(i_s,1) \vee eq(3,3) \vee \neg eq(b_s,b_r)\rangle\\ &= \langle d_s, \mathsf{T}\rangle.\end{aligned}$$

Note that $\phi(d,b_0,1,b_0,1) = \langle d, \mathsf{T}\rangle$. So by Theorem 1 and Lemma 2,

$$\tau_I(Sys(d,b_0,1,1,d,b_0,1,d,b_1,1,b_1,1)) \leftrightarrow_b B(d,\mathsf{T}).$$

Acknowledgments Jan Friso Groote is thanked for valuable discussions.

References

[1] T. Arts and I.A. van Langevelde. Correct performance of transaction capabilities. In *Proc. 2nd Conference on Application of Concurrency to System Design*, pp. 35–42. IEEE Computer Society, June 2001.

[2] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. On the consistency of Koomen's fair abstraction rule. *Theoretical Computer Science*, 51:129–176, 1987.

[3] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[4] T. Basten. Branching bisimilarity is an equivalence indeed! *Information Processing Letters*, 58:141–147, 1996.

[5] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[6] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In *Proc. 5th Conference on Concurrency Theory*, LNCS 836, pp. 401–416. Springer, 1994.

[7] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *Proc. 13th Conference on Computer Aided Verification*, LNCS 2102, pp. 250–254. Springer, 2001.

[8] K.M. Chandy and J. Misra. *Parallel Program Design. A Foundation*. Addison Wesley, 1988.

[9] A. Cimatti, F. Giunchiglia, P. Pecchiari, B. Pietra, J. Profeta, D. Romano, P. Traverso, and B. Yu. A provably correct embedded verifier for the certification of safety critical software. In *Proc. 9th Conference on Computer Aided Verification*, LNCS 1254, pp. 202–213. Springer, 1997.

[10] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.

[11] B. Courcelle. Recursive applicative program schemes. In *Handbook of Theoretical Computer Science, Volume B, Formal Methods and Semantics*, pp. 459–492. Elsevier, 1990.

[12] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In *Proc. 8th Conference on Computer-Aided Verification*, LNCS 1102, pp. 437–440. Springer, 1997.

[13] W.J. Fokkink, J.F. Groote, and J. Pang. Verification of a sliding window protocol in μCRL. In preparation.

[14] W.J. Fokkink and J.C. van de Pol. Simulation as a correct transformation of rewrite systems. In *Proceedings of 22nd Symposium on Mathematical Foundations of Computer Science*, LNCS 1295, pp. 249–258. Springer, 1997.

[15] L.-Å. Fredlund, J.F. Groote, and H.P. Korver. Formal verification of a leader election protocol in process algebra. *Theoretical Computer Science*, 177:459–486, 1997.

[16] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43:555–600, 1996.

[17] W. Goerigk and F. Simon. Towards rigorous compiler implementation verification. In *Collaboration between Human and Artificial Societies, Coordination and Agent-Based Distributed Computing*, LNCS 1624, pp. 62–73. Springer, 1999.

[18] J. F. Groote, J. Pang, and A.G. Wouters. Analysis of a distributed system for lifting trucks. *Journal of Logic and Algebraic Programming*, 2003. To appear.

[19] J. F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, 48:39–72, 2001.

[20] J.F. Groote, F. Monin, and J.C. van de Pol. Checking verifications of protocols and distributed systems by computer. In *Proc. 9th Conference on Concurrency Theory*, LNCS 1466, pp. 629–655. Springer, 1998.

[21] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Proc. 1st Workshop on the Algebra of Communicating Processes*, Workshops in Computing Series, pp. 26–62. Springer, 1995.

[22] J.F. Groote and J. Springintveld. Focus points and convergent process operators. A proof strategy for protocol verification. *Journal of Logic and Algebraic Programming*, 49:31–60, 2001.

[23] J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proc. 17th Colloquium on Automata, Languages and Programming*, LNCS 443, pp. 626–638. Springer, 1990.

[24] J.F. Groote and J.J. van Wamel. The parallel composition of uniform processes with data. *Theoretical Computer Science*, 266:631–652, 2001.

[25] B. Jonsson. *Compositional Verification of Distributed Systems.* PhD thesis, Department of Computer Science, Uppsala University, 1987.

[26] C.P.J. Koymans and J.C. Mulder. A modular approach to protocol verification using process algebra. In *Applications of Process Algebra*, Cambridge Tracts in Theoretical Computer Science 17, pp. 261–306. Cambridge University Press, 1990.

[27] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types.* Wiley/Teubner, 1996.

[28] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symposium on Principles of Distributed Computing*, pp. 137–151. ACM, 1987.

[29] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations. Part I: Untimed systems. *Information and Computation*, 121:214–233, 1995.

[30] G. Necula. Translation validation for an optimizing compiler. In *Proc. 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. SIGPLAN Notices 35:83–94. ACM, 2000.

[31] J. Pang. Analysis of a security protocol in μCRL. In *Proc. 4th International Conference on Formal Engineering Methods*, LNCS 2495, pp. 396–400. Springer, 2002.

[32] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. 4th Conference on Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1384, pp. 151–166. Springer, 1998.

[33] J.C. van de Pol and M. Valero Espada. Formal specification of Javaspaces^{TM} architecture using μCRL. In *Proc. 5th Conference on Coordination Models and Languages*, LNCS 2315, pp. 274–290. Springer, 2002.

[34] C. Shankland and M.B. van der Zwaag. The tree identify protocol of IEEE 1394 in μCRL. *Formal Aspects of Computing*, 10:509–531, 1998.

[35] Y.S. Usenko. Linearization of μCRL specifications (extended abstract). In *Proc. 3rd International Workshop on Verification and Computational Logic (VCL2002)*, Technical Report DSSE-TR-2002-5. Department of Electronics and Computer Science, University of Southampton, 2002.

[36] M.B. van der Zwaag. The cones and foci proof technique for timed transition systems. *Information Processing Letters*, 80(1):33–40, 2001.

Towards a Behavioural Theory of Access and Mobility Control in Distributed Systems
(Extended Abstract)

Matthew Hennessy[1], Massimo Merro[2], and Julian Rathke[1]

[1] School of Cognitive and Computing Sciences
University of Sussex
matthewh,julianr@cogs.sussex.ac.uk
[2] Dipartimento di Informatica
Universitá di Verona
merro@sci.univr.it

Abstract. We define a typed bisimulation equivalence for the language Dpi, a distributed version of the π-calculus in which processes may migrate between dynamically created locations. It takes into account resource access policies, which can be implemented in Dpi using a novel form of dynamic capability types. The equivalence, based on typed actions between configurations, is justified by showing that it is *fully-abstract* with respect to a natural distributed version of a contextual equivalence.
In the second part of the paper we study the effect of controlling the migration of processes. This affects the ability to perform observations at specific locations, as the observer may be denied access. We show how the typed actions can be modified to take this into account, and generalise the *full-abstraction* result to this more delicate scenario.

1 Introduction

The behaviour of processes in a distributed system depends on the resources they have been allocated. Moreover these resources, or a process's knowledge of these resources, may vary over time. Therefore an adequate behavioural theory of distributed systems must be based not only on the inherent abilities of processes to interact with other processes, but must also take into account the (dynamic) resource environment in which they are operating. In our approach judgements will take the form

$$\Gamma \models M \approx N,$$

where N and M are systems and Γ represents their computing environment. Intuitively this means that M and N offer the same behaviour, relative to the environment Γ. The challenge addressed by this paper is to give an adequate formalisation of this idea, where

– the systems M and N are collections of *location aware* processes, which may be allocated varying *access rights* to resources at different locations and may migrate between these locations to exercise their rights

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 282–298, 2003.

– the computing environment Γ may vary dynamically, reflecting both the overall resources available to M and N and the evolving knowledge that users may accumulate of these resources.

This is developed in terms of the language Dpi, [9], a version of the π-calculus, [13], in which processes may migrate between locations, which in turn can be dynamically created. As explained in [9] resource access policies in Dpi may be implemented using a *capability* based type system; thus in this setting it is sufficient to develop typed behavioural equivalences in order to capture the effect of resource access policies on process behaviour. But in this paper we extend the typing system of [9] by allowing types to be created *dynamically* and to depend on received data.

The language Dpi and its reduction semantics is summarised in Section 2, which relies heavily on the corresponding section of [9]. The typing system, discussed briefly in Section 3, contains two major extensions to the original typing system of [9]. The first introduces a new kind of type, $\mathsf{rc}\langle \mathrm{A} \rangle$ for *registered channel names*, which allows channels names to be used consistently at multiple locations. The second allows types to be constructed *dynamically* and be dependent on received data. Subject reduction still holds for this extended type system.

The second novelty of the paper, in Section 4.1, is a definition of *typed action*

$$\Gamma \rhd M \xrightarrow{\mu} \Gamma' \rhd N \tag{1}$$

indicating that in an environment constrained by the *type environment* Γ the system M may perform the action μ and be transformed into N; the environment Γ may also be changed by this interaction, to Γ', for example by the extrusion of new resources, or new capabilities on already known resources. Here the actions μ are either internal moves, τ, or *located* input or output actions, of the form $k.a?v$ or $(\tilde{b})k.a!v$. Informally, the action in (1) is possible if M is capable of performing the action μ in the standard manner *and* the environment Γ allows it to happen.

With these typed actions we can define a standard notion of (weak) bisimulations between *configurations*, consistent pairs of the form $\Gamma \rhd M$; the formal definition is given in Section 4 and we use $\Gamma \rhd M \approx^{bis} N$ to denote that there is a bisimulation containing the two configurations $\Gamma \rhd M$ and $\Gamma \rhd N$.

The first main result of the paper is that this notion of *typed bisimulation* captures precisely an independently defined *contextual equivalence*, which we denote by $\eqsim^{rbc}$.

The final topic of the paper is the effect of *migration* on the behaviour of systems. In Dpi the migration of processes is unconstrained. The relevant reduction rule is

$$k[\![\mathsf{goto}\, l.P]\!] \to l[\![P]\!].$$

Any agent is allowed to migrate from a site k to the site l. Indeed this rule is essential in establishing the above theorem. For example consider the systems $M_1,\ M_2$ given by

$$(\mathsf{new}\, k : \mathrm{K})\ l[\![c!\langle k\rangle]\!] \mid k[\![a!\langle\rangle\, \mathsf{stop}]\!] \qquad \text{and} \qquad (\mathsf{new}\, k : \mathrm{K})\ l[\![c!\langle k\rangle]\!] \mid k[\![\mathsf{stop}]\!] \tag{2}$$

where K is the declaration type $\mathsf{loc}[a : \mathsf{rw}\langle\rangle]$, and Γ an environment which has read capability at type K on c at l. These are not bisimilar in the environment Γ, as after exporting the new name k on c at l, that is performing the *bound* output action $(k)l.c!k$, only the former may have the typed action

$$(\Gamma' \rhd k[\![a!\langle\rangle\,\mathsf{stop}]\!]) \xrightarrow{k.a!\langle\rangle} (\Gamma' \rhd k[\![\mathsf{stop}]\!])$$

where Γ' represents the environment Γ updated with the new knowledge of a at k. Moreover they can be distinguished contextually because of the Γ-context

$$l[\![c?(x)\,.\mathsf{goto}\,x.a?(y)\,\mathsf{goto}\,l.\omega!\langle\rangle]\!] \mid -$$

An environment which has read or write capability on a channel at k can automatically send an agent there to perform a test and report back to base. Note that this test works only because systems allowed by Γ have the automatic ability to migrate to the site k. If on the other hand migration were constrained, as one would expect in more realistic scenarios, then these tests would no longer be necessarily valid and these terms may become contextually equivalent.

There are many mechanisms by which migration could be controlled in languages such as Dpi. In this paper we introduce one such mechanism, based on a simple extension of the typing system, which allows us to examine the effect of such control on behavioural equivalences. We introduce a new location capability **move**. Then migration from any location to k is only allowed with respect to an environment Γ, if in Γ the location k is known with a capability **move**; we say Γ has *migration* rights to k. This idea is easily implemented by using a slight extension to our typing system, and is sufficient to demonstrate the subtleties involved when migration is controlled.

The remainder of the paper is devoted to extending the characterisation result given above, characterising a contextual equivalence using bisimulations, to this language. The typed actions (1) above are readily adapted to this scenario. Here we allow, for example, the action

$$\Gamma \rhd M \xrightarrow{k.a!v}_m \Gamma' \rhd N \tag{3}$$

if, in addition to the requirements for (1), the environment has the capability **move** for k; intuitively for the environment to see the action (3) it must be able to move to the site k.

These actions lead to a new bisimulation equivalence, denoted $\approx^m_{bis}$, and we can prove

$$\Gamma \models M \approx^m_{bis} N \text{ if and only if } \Gamma \models M \mathrel{\overline{\approx}}^m_{cxt} N$$

where $\mathrel{\overline{\approx}}^m_{cxt}$ is a suitable modification of the contextual equivalence $\mathrel{\overline{\approx}}^{rbc}$; in the definition of $\mathrel{\overline{\approx}}^m_{cxt}$ we only require the equivalence to be preserved in the context of processes at locations to which the environment has migration rights. Thus referring to (2) above we will have

$$\Gamma \models M_1 \mathrel{\overline{\approx}}^m_{cxt} M_2$$

if Γ does not have migration rights to k. Note that neither of these systems can give rise to the modified typed actions, as given in (3) above.

However it is easy to envisage a natural version of contextual equivalence which does distinguish between M_1 and M_2 of (2) above. Although the environment may not have migration rights to k, it may, apriori, have a process running there. If this were allowed, and the environment had the appropriate capability on the channel a at k then the systems M_1 and M_2 could be distinguished. The question then arises of finding a bisimulation based characterisation for this modified contextual equivalence.

We address a parameterised version of this problem. Let $\mathcal{T}$ be the set of locations at which apriori the environment can place testing processes and let $\eqsim_{cxt}^{\mathcal{T}}$ be the resulting contextual equivalence. Unfortunately this is not characterised by the natural modification to the equivalence $\approx_{bis}^{m}$, which we denote by $\approx_{bis}^{\mathcal{T}}$. A counterexample is given in Section 5.2.

It turns out that we must be careful about the location at which information is learnt. Information about k learnt at l can not be used without the capability to move to k. However this information must be retained because that move capability may subsequently be obtained. This leads to a more complicated form of environment $\overline{\Gamma}$, which records

- locations at which testing processes may be placed, $\mathcal{T}$
- *globally* available information on capabilities at locations
- similar *locally* available information.

The details are given in Section 5.2, which also contains a generalisation of the typed actions of (1) above to these more complicated environments. The final result of the paper is that the new bisimulation equivalence based on these actions again captures the contextual equivalence.

In this extended abstract all proofs are omitted, as indeed is most of the technical exposition of the typing system. The reader is referred to the full version of the paper, [7], for all the details.

2 The Language Dpi

Syntax: The syntax, given in Figure 1, is a slight extension of that of Dpi from [9]. This presupposes a general set of names *Names* ranged over by n, m, and a set of variables *Vars* ranged over by x, y; informally we will often use $a, b, c, \ldots$ for names of channels and $l, k, \ldots$ for locations or sites. *Identifiers*, ranged over by u, v, w, may either be variables or names. The syntax also uses a set of types; discussion of these is postponed until the next section.

The syntax is built in a two-level structure, the lower level being processes, agents or threads. The syntax here is an extension of the π-calculus, [9], with primitives for migration between locations. At this level there are three forms of name creation:

Fig. 1. Syntax of DPI

$M, N ::=$	*Systems*
$l[\![P]\!]$	Located Process
$M \mid N$	Composition
$(\mathsf{new}\, n : \mathrm{T})\ M$	Name Scoping
$\mathbf{0}$	Termination

$P, Q ::=$	*Processes*
$u!\langle V\rangle\, P$	Output
$u?(X : \mathrm{T})P$	Input
$\mathsf{goto}\, v.W$	Migration
$\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q$	Matching
$(\mathsf{newc}\, n : \mathrm{A})\ P$	Channel Name creation
$(\mathsf{newreg}\, n : \mathrm{G})\ P$	Registered Name creation
$(\mathsf{newloc}\, k : \mathrm{K})\ \mathsf{with}\ C\ \mathsf{in}\ P$	Location Name creation
$P \mid Q$	Composition
$* P$	Replication
stop	Termination

$U, V, W ::=$	*Values*
$(\alpha_1, \ldots, \alpha_n),\ n > 0$	tuples

$\alpha, \alpha' ::=$	*Generalised Identifiers*
u	Identifiers
$(u_1, \ldots, u_n)@u,\ n \geq 0$	Located Identifiers

- $(\mathsf{newc}\, a : \mathrm{A})\ P$, the creation of a new *local channel name* of type A called a.
- $(\mathsf{newreg}\, n : \mathsf{rc}\langle \mathrm{A}\rangle)\ P$, the creation of a new *registered name* for located channels of type A. These may be used in the declaration types of locations and treated uniformly across them.
- $(\mathsf{newloc}\, k : \mathrm{K})\ \mathsf{with}\ C\ \mathsf{in}\ P$, the creation of a new *location name* k of type K, with the code C running there, and P as a local continuation. Our typing system will ensure that K is a well-formed location type; for example this means that it may only use the registered channel names.

Systems are constructed from *located threads*, of the form $l[\![P]\!]$, representing the thread P running at location l. These may be combined with the parallel operator $\mid$ and names may be shared between threads using the construct $(\mathsf{new}\, e : \mathrm{T})$.

Reduction Semantics: This is given in terms of a binary relation between *closed* systems, system terms which have no free occurrences of variables:

$$M \to N$$

Fig. 2. Reduction semantics for Dpi

$$\frac{}{k[\![c!\langle V\rangle\, Q]\!] \mid k[\![c?(X:\mathrm{T})\, P]\!] \to k[\![Q]\!] \mid k[\![P\{^{V}/_{X}\}]\!]}\ (\text{R-COMM}) \qquad \frac{}{k[\![P \mid Q]\!] \to k[\![P]\!] \mid k[\![Q]\!]}\ (\text{R-SPLIT})$$

$$\frac{}{k[\![(\mathsf{newc}\, n:\mathrm{A})\ P]\!] \to (\mathsf{new}\, n:\mathrm{A}@k)\ k[\![P]\!]}\ (\text{R-C-CREATE}) \qquad \frac{}{k[\![\mathsf{goto}\, l.P]\!] \to l[\![P]\!]}\ (\text{R-MOVE})$$

$$\frac{}{k[\![(\mathsf{newreg}\, n:\mathrm{G})\ P]\!] \to (\mathsf{new}\, n:\mathrm{G})\ k[\![P]\!]}\ (\text{R-R-CREATE}) \qquad \frac{}{k[\![(\mathsf{newloc}\, l:\mathrm{L})\ \mathsf{with}\ C\ \mathsf{in}\ P]\!] \to (\mathsf{new}\, l:\mathrm{L})\ (l[\![C]\!] \mid k[\![P]\!])}\ (\text{R-L-CREATE})$$

$$\frac{}{k[\![*\, P]\!] \to k[\![P \mid *\, P]\!]}\ (\text{R-UNWIND}) \qquad \frac{}{k[\![\mathsf{if}\ u = u\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \to k[\![P]\!]}\ (\text{R-EQ})$$

$$\frac{M \equiv N,\ M \to M',\ M' \equiv N'}{N \to N'}\ (\text{R-STR}) \qquad \frac{u \neq v}{k[\![\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \to k[\![Q]\!]}\ (\text{R-NEQ})$$

Fig. 3. Structural equivalence for Dpi

$$\begin{aligned}
&(\text{S-EXTR}) && (\mathsf{new}\, n:\mathrm{T})\ (M \mid N) = M \mid (\mathsf{new}\, n:\mathrm{T})\ N \quad \text{if } \mathsf{n}(n) \notin \mathsf{fn}(M)\\
&(\text{S-COM}) && M \mid N = N \mid M\\
&(\text{S-ASSOC}) && (M \mid N) \mid O = M \mid (N \mid O)\\
&(\text{S-ZERO}) && M \mid \mathbf{0} = M\\
&(\text{S-FLIP}) && (\mathsf{new}\, n:\mathrm{T})\ (\mathsf{new}\, n':\mathrm{T}')\ N = (\mathsf{new}\, n':\mathrm{T}')\ (\mathsf{new}\, n:\mathrm{T})\ N \quad \text{if } \mathsf{n}(n) \notin \mathrm{T}', \mathsf{n}(n') \notin \mathrm{T}
\end{aligned}$$

and is a mild generalisation of that given in [9] for Dpi. It is a *contextual relation* between systems; that is, it is preserved by the static operators $\mid$ and $(\mathsf{new}\, e:\mathrm{E})$. The defining rules are given in Figure 2, one of which uses a structural equivalence, whose axioms are given in Figure 3.

3 Typing

The collection of types, given in Figure 4, is an extension of those used in [9], to which the reader is referred for more background and motivation. Note that the formation of local channel type $\mathsf{rw}\langle \mathrm{T}, \mathrm{U}\rangle$ requires a subtyping relation $<:$; however in the examples of this extended abstract we only use types of the form

Fig. 4. Types

Base Types:	$\mathrm{B} ::= \mathbf{int} \mid \mathbf{bool} \mid \mathbf{unit} \mid \top \mid \ldots$
Local Channel types:	$\mathrm{A} ::= \mathsf{r}\langle \mathrm{T} \rangle \mid \mathsf{w}\langle \mathrm{T} \rangle \mid \mathsf{rw}\langle \mathrm{T}, \mathrm{U} \rangle$ provided $\mathrm{U} <: \mathrm{T}$
Capability Types:	$\mathrm{R} ::= u : \mathrm{A}$
Location Types:	$\mathrm{K} ::= \mathsf{loc}[\mathrm{R}_1, \ldots, \mathrm{R}_n], n \geq 0$
Registered Name Types:	$\mathrm{G} ::= \mathsf{rc}\langle \mathrm{A} \rangle$
Value Types:	$\mathrm{C} ::= \mathrm{B} \mid \mathrm{A} \mid G \mid (\tilde{\mathrm{A}})_{@}u \mid (\tilde{\mathrm{A}})_{@}\mathrm{K}$
Transmission Types:	$\mathrm{T} ::= (\mathrm{C}_1, \ldots, \mathrm{C}_n), n \geq 0$

$\mathsf{rw}\langle \mathrm{T}, \mathrm{T} \rangle$, which we abbreviate to $\mathsf{rw}\langle \mathrm{T} \rangle$. But there are four major differences from [9]:

- We use a *top* type $\top$ for names with no capabilities.
- We use a new category of types, *registered name types*, to explicitly manage the resource names which can be shared between different locations.
- The types expressions are allowed to contain variables, thereby giving rise to what we call *dynamic* types; the constraints they place on agent behaviour is determined dynamically by instantiation of these variables.
- The notion of type environment is changed; they do not explicitly contain associations between names and location types.

A type judgement will take the form $\Gamma \vdash M$ where Γ is a *type environment*, a list of assumptions about the types to be associated with the identifiers in the system M. Typical examples used in environments include $w : \mathsf{rw}\langle \mathrm{T} \rangle_{@}k$, meaning w is a read/write channel at location k, $w : \mathsf{rc}\langle \mathrm{A} \rangle$ meaning w is a registered channel name, and $w : \mathsf{loc}$ meaning that w is a location name. Because of lack of space in this extended abstract we omit the rules for the judgements; they may be found in the full version, [7], which also contains a proof of Subject Reduction.

4 Contextual Equivalence in DPI

We now turn to the issue of defining a notion of equivalence for our language. In general the ability to distinguish between systems depends on our knowledge of the capabilities at the various sites. For example a client who is not aware of the resource a at the location k will be unable to perceive any difference between the two systems $k[\![a?(x)\, P]\!]$ and $k[\![\mathsf{stop}]\!]$. Thus, as explained in the Introduction, we develop notions of equivalences of the form

$$\Gamma \models M \approx N \tag{4}$$

where Γ is a well-defined type environment representing the user's knowledge of the capabilities of the systems M and N. It is important to realise that Γ may not contain enough information to type M, N; it represents that part of the type environment of these processes which has been released to the user.

A *knowledge-indexed relation over systems* is a family of binary relations between closed systems indexed by closed type environments. We write $\Gamma \models M \mathcal{R} N$ to mean that systems M and N are related by $\mathcal{R}$ at index Γ; moreover we assume that both M and N are typeable relative to some *extension* of Γ; that is some environment Δ with the same domain as Γ such that $\Delta <: \Gamma$. The desirable properties of knowledge-indexed relations which we consider are as follows:

Reduction closure: We say that a knowledge-indexed relation over systems is *reduction closed* if whenever $\Gamma \models M \mathcal{R} N$ and $M \to M'$ there exists some N' such that $N \to^* N'$ and $\Gamma \models M \mathcal{R} N'$.

Context closure: We say that a knowledge-indexed relation over systems is *contextual* if

(i) $\Gamma \models M \mathcal{R} N$ implies $\Gamma, \Gamma' \models M \mathcal{R} N$
(ii) $\Gamma \models M \mathcal{R} N$ and $\Gamma \vdash O$ implies $\Gamma \models (M \mid O) \mathcal{R} (N \mid O)$
(iii) $\Gamma, n : \mathrm{T} \models M \mathcal{R} N$ implies $\Gamma \models (\mathsf{new}\, n : \mathrm{T})\ M\ \mathcal{R}\ (\mathsf{new}\, n : \mathrm{T})\ N$

Note that in this last clause we have used an abbreviation to cover the three different forms of names which can be declared, local channels, registered names and locations, each differentiated by the form which T can take. Moreover we assume that n is new to Γ.

Barb preservation: For any given location k and any given channel a such that $\Gamma \vdash k : \mathsf{loc}$ and $\Gamma \vdash a : \mathsf{rw}\langle\rangle_{@}k$ we write $\Gamma \vdash M \Downarrow^{\mathsf{barb}} a_{@}k$ if there exists some M' such that $M \to^* (\mathsf{new}\, \tilde{n})\ (M' \mid k[\![a!\langle\rangle\, P]\!])$, where k, a do not appear in $\tilde{n}$. We say that a knowledge-indexed relation over systems is *barb preserving* if

$$\Gamma \models M \mathcal{R} N \quad \text{and} \quad \Gamma \vdash M \Downarrow^{\mathsf{barb}} a_{@}k \quad \text{implies} \quad \Gamma \vdash N \Downarrow^{\mathsf{barb}} a_{@}k$$

These three properties determine our *touchstone* equivalence:

Definition 1 (Reduction barbed congruence). *We let* $\mathrel{\widetilde{\approx}}^{rbc}$ *be the largest knowledge-indexed relation over systems which is*

- *pointwise symmetric, that is* $\Gamma \models M \mathrel{\widetilde{\approx}}^{rbc} N$ *implies* $\Gamma \models N \mathrel{\widetilde{\approx}}^{rbc} M$
- *contextual*
- *reduction closed*
- *barb preserving* □

We will now characterise $\mathrel{\widetilde{\approx}}^{rbc}$ using a labelled transition system and bisimulation equivalence, thereby justifying our particular notion of bisimulations.

Fig. 5. Typed actions

(LTS-RED)
$$\frac{M \longrightarrow M'}{(\Gamma \rhd M) \xrightarrow{\tau} (\Gamma \rhd M')}$$

(LTS-OUT)
$$\frac{\begin{array}{l}\Gamma \vdash k : \mathsf{loc} \\ a : \mathsf{r}\langle \mathrm{T}\rangle_{@k} \in \Gamma \\ \Gamma \sqcap \langle V : \mathrm{T}\rangle_{@k} \text{ exists}\end{array}}{(\Gamma \rhd k[\![a!\langle V\rangle\, P]\!]) \xrightarrow{k.a!V} (\Gamma \sqcap \langle V : \mathrm{T}\rangle_{@k} \rhd k[\![P]\!])}$$

(LTS-IN)
$$\frac{\begin{array}{l}\Gamma \vdash k : \mathsf{loc} \\ a : \mathsf{w}\langle \mathrm{U}\rangle_{@k} \in \Gamma \\ \Gamma \vdash V : \mathrm{U}_{@k}\end{array}}{(\Gamma \rhd k[\![a?(X : \mathrm{T})\, P]\!]) \xrightarrow{k.a?V} (\Gamma \rhd k[\![P\{\!\{{}^{V}/_{X}\}\!\}]\!])}$$

(LTS-OPEN)
$$\frac{(\Gamma, n : \top \rhd M) \xrightarrow{(\tilde{n})k.a!V} (\Gamma' \rhd M') \quad n \neq a, k}{(\Gamma \rhd (\mathsf{new}\, n : \mathrm{T})\ M) \xrightarrow{(n\tilde{n})k.a!V} (\Gamma' \rhd M') \quad n \in \mathsf{fn}(V)}$$

(LTS-WEAK)
$$\frac{(\Gamma, n : T \rhd M) \xrightarrow{(\tilde{n}:\tilde{\mathrm{T}})k.a?V} (\Gamma' \rhd M') \quad n \neq a, k}{(\Gamma \rhd M) \xrightarrow{(n:\mathrm{T}\tilde{n}:\tilde{\mathrm{T}})k.a?V} (\Gamma' \rhd M')}$$

(LTS-PAR)
$$\frac{(\Gamma \rhd M) \xrightarrow{\alpha} (\Gamma' \rhd M') \quad \mathsf{bn}(\alpha) \notin \mathsf{fn}(N)}{\begin{array}{l}(\Gamma \rhd M \mid N) \xrightarrow{\alpha} (\Gamma' \rhd M' \mid N) \\ (\Gamma \rhd N \mid M) \xrightarrow{\alpha} (\Gamma' \rhd N \mid M')\end{array}}$$

(LTS-NEW)
$$\frac{(\Gamma, n : \top \rhd M) \xrightarrow{\alpha} (\Gamma', n : \top \rhd M') \quad n \notin \mathsf{n}(\alpha)}{(\Gamma \rhd (\mathsf{new}\, n : \mathrm{T})\ M) \xrightarrow{\alpha} (\Gamma' \rhd (\mathsf{new}\, n : \mathrm{T})\ M')}$$

4.1 A Labelled Transition Characterisation of Contextual Equivalence

A standard labelled transition system for DPI would describe the actions, inputs/outputs on located channels, which a system could in principle perform. However because of possible limited knowledge an external user may not be able to provoke these actions. Our labelled transition system uses *typed actions* of the form

$$\Gamma \rhd M \xrightarrow{\mu} \Gamma' \rhd M'$$

and the defining rules are given in Figure 5. Again the intuition here is that M is not necessarily typeable by Γ; instead it represents that part of the environment which does type M which is known by the user.

As an example the rule (LTS-OUT) says that $k[\![a!\langle V\rangle\, P]\!]$ can only perform the obvious output action if

- k is known by Γ to be a location
- the user has the capability to accept a value from a at k, that is $\Gamma \vdash a : \mathsf{r}\langle \mathrm{T}\rangle_@k$ for some transmission type T
- the information which is being sent to the user does not contradict its current knowledge. This is formalised using a partial meet operation $\sqcap$ which is defined directly on types and extended to type environments; it is a method for *consistently* combining type information. In the the rule we also use the notation $\langle V : \mathrm{T}\rangle_@k$ to represent a simple environment obtained by adding the knowledge that V has type T at location k. The formal definition may be found in [7].

The rule for input, (LTS-IN), has a similar flavour.

Definition 2 (Bisimulations). *A binary relation $\mathcal{R}$ is said to be a* bisimulation *if $\Gamma \rhd M\ \mathcal{R}\ \Gamma \rhd N$ implies*

- *$\Gamma \rhd M \xrightarrow{\mu} \Gamma' \rhd M'$ implies $\Gamma \rhd N \stackrel{\hat{\mu}}{\Longrightarrow} \Gamma' \rhd N'$ for some N' such that $\Gamma' \rhd M'\ \mathcal{R}\ \Gamma' \rhd N'$*
- *Symmetrically, $\Gamma \rhd N \xrightarrow{\mu} \Gamma' \rhd N'$ implies $\Gamma \rhd M \stackrel{\hat{\mu}}{\Longrightarrow} \Gamma' \rhd M'$ for some N' such that $\Gamma' \rhd M'\ \mathcal{R}\ \Gamma' \rhd N'$*

Here we are using the standard notation from [12]; $\stackrel{\mu}{\Longrightarrow}$ means $\xrightarrow{\tau}{}^* \circ \xrightarrow{\mu} \circ \xrightarrow{\tau}{}^*$ while $\stackrel{\hat{\mu}}{\Longrightarrow}$ is $\xrightarrow{\tau}{}^*$ if μ is τ and $\stackrel{\mu}{\Longrightarrow}$ otherwise; this allows a single internal move to be matched by zero or move internal moves.

We write $\Gamma \models M \approx^{bis} N$ if $(\Gamma \rhd M)\ \mathcal{R}\ (\Gamma \rhd N)$ for some bisimulation $\mathcal{R}$, and say that M and N are bisimilar in the environment Γ.

Theorem 1 (Full abstraction of $\eqsim^{rbc}$ for $\approx^{bis}$). *$\Gamma \models M \eqsim^{rbc} N$ iff $\Gamma \models M \approx^{bis} N$* □

5 Controlling Mobility

We now consider a richer calculus in which movement of processes may be controlled. As explained in the Introduction we extend DPI with a very simple means of mobility control and investigate the resulting contextual equivalence.

5.1 Migration Rights

Hennessy and Riely have already proposed a simple access control mechanism for DPI in the form of the *move* capability [9], and here we investigate the effect this has on behavioural equivalence.

The location types in DPI are of the form $\mathsf{loc}[u_1 : \mathrm{A}_1, \ldots, u_n : \mathrm{A}_n]$ where the $u_i : \mathrm{A}_i$ can be seen as capabilities at that location. We now introduce an extra type of capability by allowing location types to be also of the form

$$\mathsf{loc}[\mathbf{move}, a_1 : \mathrm{A}_1, \ldots, a_n : \mathrm{A}_n]. \tag{5}$$

If a location k is known at this type then agents resident at any location have migration rights to k; we realise that this is a somewhat simplistic approach to mobility control but it enables us to demonstrate the subtleties involved in developing behavioural theories in the presence of any such capabilities.

The type system, and the type-checking rules, can easily modified to cater for this new capability while retaining Subject Reduction; for details see the full version of the paper, [7].

Instead let us examine the effect migration rights have on behavioural equivalences. Suppose $N_1,\ N_2$ are given by

$$k[\![a!\langle\rangle\,\mathsf{stop}]\!] \qquad \text{and} \qquad k[\![\mathsf{stop}]\!] \tag{6}$$

The question of whether or not N_1 and N_2 are contextually equivalent relative to an environment Γ, now written $\Gamma \models N_1 \mathrel{\widetilde{\approx}}^{m}_{cxt} N_2$, depends on whether Γ has migration rights to k. If so, say at a location l, agents may be sent from l to k in order to observe the difference in behaviour between N_1 and N_2 at k. But will these agents be able to report back to the environment? This in turn depends on whether the environment allows migration from k to l. In general, the situation can get more complicated. Observing different behaviour at a site k may require a range of capabilities, and knowledge of these may be distributed throughout the environment at sites with limited migration rights between themselves.

For this extended language we give, in the following two subsections, two different generalisations to the full-abstraction result, Theorem 1.

5.2 Mobility Bisimulation Equivalence

It is straightforward to adapt the typed actions in Figure 5 to take into account these simple migration rights. Essentially for an action to be allowed at a site k the constraints discussed in Section 4.1 must be satisfied but in addition the environment must have migration rights to k. Formally we define actions

$$\Gamma \rhd M \xrightarrow{\mu}_{m} \Gamma' \rhd M'$$

by replacing the rules (LTS-OUT) and (LTS-IN) in Figure 5 with

$$\frac{\begin{array}{l}\Gamma \vdash k : \mathsf{loc}[\mathbf{move}] \\ a : \mathsf{r}\langle \mathrm{T}\rangle_{@}k \in \Gamma \\ \Gamma \sqcap \langle V : \mathrm{T}\rangle_{@}k \text{ exists}\end{array}}{(\Gamma \rhd k[\![a!\langle V\rangle\, P]\!]) \xrightarrow{k.a!V}_m (\Gamma \sqcap \langle V : \mathrm{T}\rangle_{@}k \rhd k[\![P]\!])} \quad (\text{LTS-OUT}_{\mathrm{M}})$$

$$\frac{\begin{array}{l}\Gamma \vdash k : \mathsf{loc}[\mathbf{move}] \\ a : \mathsf{w}\langle \mathrm{U}\rangle_{@}k \in \Gamma \\ \Gamma \vdash V : \mathrm{U}_{@}k\end{array}}{(\Gamma \rhd k[\![a?(X : \mathrm{T})\, P]\!]) \xrightarrow{k.a?V}_m (\Gamma \rhd k[\![P\{\!\{^V\!/_X\}\!\}]\!])} \quad (\text{LTS-IN}_{\mathrm{M}})$$

and leaving the other rules unchanged.

Definition 3 (Typed m-Bisimulations). *Let $\Gamma \models M \approx^m_{bis} N$ denote the resulting version of bisimulation equivalence, obtained by replacing the use of $\Gamma \rhd M \xrightarrow{\mu} \Gamma' \rhd M'$ in Definition 2 with $\Gamma \rhd M \xrightarrow{\mu}_m \Gamma' \rhd M'$.*

Example 1. As in (6) above let N_1, N_2 denote $k[\![a!\langle\rangle\,\mathsf{stop}]\!]$ and $k[\![\mathsf{stop}]\!]$ respectively, and suppose Γ is such that $\Gamma \not\vdash k : \mathsf{loc}[\mathbf{move}]$. Then $\Gamma \models N_1 \approx^m_{bis} N_2$ because no m-typed actions are possible from these systems.

Example 2. Let N_3, N_4 represent $(\mathsf{new}\, k : \mathsf{loc}[\mathbf{move}, b : \mathsf{rw}\langle\rangle])\ l[\![a!\langle k\rangle]\!] \mid k[\![b!\langle\rangle]\!]$ and $(\mathsf{new}\, k : \mathsf{loc}[\mathbf{move}, b : \mathsf{rw}\langle\rangle])\ l[\![a!\langle k\rangle]\!] \mid k[\![\mathbf{0}]\!]$ respectively, and let Γ_1 denote the environment

$$l : \mathsf{loc},\ l : \mathbf{move},\ b : \mathsf{rc}\langle\mathsf{rw}\langle\rangle\rangle,\ a : \mathsf{rw}\langle\mathsf{loc}[b : \mathsf{rw}\langle\rangle]\rangle_{@}l$$

Here the environment can interact at the site l because it has migration rights there; and via the channel a located at l it can gain knowledge of k. But because of the type at which it knows a it can never gain migration rights to k. Consequently we have $\Gamma_1 \models N_3 \approx^m_{bis} N_4$.

However let Γ_2 denote

$$l : \mathsf{loc},\ l : \mathbf{move},\ b : \mathsf{rc}\langle\mathsf{rw}\langle\rangle\rangle,\ a : \mathsf{rw}\langle\mathsf{loc}[\mathbf{move}, b : \mathsf{rw}\langle\rangle]\rangle_{@}l$$

Here any location name received on the channel a at l comes with migration rights. So we have $\Gamma_2 \models N_3 \not\approx^m_{bis} N_4$.

The essential property of this new form of equivalence is a restricted form of contextuality:

Proposition 1. *Suppose $\Gamma \vdash k : \mathsf{loc}[\mathbf{move}]$. Then $\Gamma \models M \approx^m_{bis} N$ and $\Gamma \vdash k[\![P]\!]$ implies $\Gamma \models M \mid k[\![P]\!] \approx^m_{bis} N \mid k[\![P]\!]$.*

This property allows us to give a contextual characterisation of $\approx^m_{bis}$. We need to slightly adapt the concepts defined in Section 4.

m-Context closure: Here the change is in the second clause of the definition of *Context closure*; it is changed to

(ii) $\Gamma \models M \mathcal{R} N$, $\Gamma \vdash k : \mathsf{loc}[\mathbf{move}]$ and $\Gamma \vdash k[\![P]\!]$ implies $\Gamma \models (M \mid k[\![P]\!]) \mathcal{R} (N \mid k[\![P]\!])$

m-Barb preservation: Here we only allow barbs at locations to which migration rights exist. We write $\Gamma \vdash M \Downarrow^{\mathsf{mbarb}} a@k$ if in addition to the requirement that $\Gamma \vdash M \Downarrow^{\mathsf{barb}} a@k$ we have $\Gamma \vdash k : \mathsf{loc}[\mathbf{move}]$ and $\Gamma \vdash a : \mathsf{rw}\langle\rangle@k$.

We now say that a typed relation over systems is *m-barb preserving* if $\Gamma \models M \mathcal{R} N$ and $\Gamma \vdash M \Downarrow^{\mathsf{mbarb}} a@k$ implies $\Gamma \vdash N \Downarrow^{\mathsf{mbarb}} a@k$.

Definition 4 (m-Reduction barbed congruence). *Let $\cong^m_{cxt}$ be the largest typed relation over systems which is reduction-closed, m-contextual and m-barb preserving.*

Our first generalisation may now be stated:

Theorem 2 (Full abstraction of $\cong^m_{cxt}$ for $\approx^m_{bis}$). *$\Gamma \models M \cong^m_{cxt} N$ iff $\Gamma \models M \approx^m_{bis} N$.* □

5.3 Re-examining Contextuality

The two examples given in the previous subsection deserve re-examination, particularly in view of the definition of m-contextuality. In Example 1 above it turns out that N_1 and N_2 are not equivalent with respect to any Γ which does not contain migration rights to k. But an alternative definition of *contextual* would require the behavioural equivalence to be preserved by *all* contexts typeable by Γ. Suppose Γ is the environment

$$h : \mathsf{loc},\ h : \mathbf{move},\ \mathit{eureka} : \mathsf{rw}\langle\rangle@h,\ k : \mathsf{loc},\ a : \mathsf{rw}\langle\rangle@k$$

Then one can check that $\Gamma \vdash k[\![a?() \,\mathsf{goto}\, h.\mathit{eureka}!\langle\rangle]\!]$ and running N_i in parallel with this well-typed context would enable us to distinguish between them.

This new, but still informal, notion of contextuality presupposes that the context can have already in place some testing agents running at certain sites to which it does not have migration rights. An obvious choice of sites would be all those which are known about, that is all k such that $\Gamma \vdash k : \mathsf{loc}$. However our results can be parameterised on this choice.

$\mathcal{T}$-Context closure: Let $\mathcal{T}$ be a collection of location names. The concept of $\mathcal{T}$-Context closure is obtained by changing the second clause in the definition of *Context closure* to:

(ii) $\Gamma \models M \mathcal{R} N$, $\Gamma \vdash k[\![P]\!]$, where either $k \in \mathcal{T}$ or $\Gamma \vdash k : \mathsf{loc}[\mathbf{move}]$, implies $\Gamma \models (M \mid k[\![P]\!]) \mathcal{R} (N \mid k[\![P]\!])$

Definition 5 ($\mathcal{T}$-Reduction barbed congruence). *Let $\cong^{\mathcal{T}}_{cxt}$ be the largest typed relation over systems which is reduction-closed, $\mathcal{T}$-contextual and m-barb preserving.*

The question now is whether we can devise a bisimulation based characterisation of $\cong^{\mathcal{T}}_{cxt}$.

The obvious approach is to modify the definitions of the typed actions $\xrightarrow{\mu}_m$, to obtain actions $\xrightarrow{\mu}_{\mathcal{T}}$ which allow observations at a site k, if either the environment has migration rights to k as before, **or** $k \in \mathcal{T}$. With these actions we can modify Definition 2 to obtain a new behavioural equivalence, which we denote by $\approx^{\mathcal{T}}_{bis}$. Unfortunately this does not coincide with the contextual equivalence $\cong^{\mathcal{T}}_{cxt}$.

Example 3. Let $N_5,\ N_6$ be the systems defined by

$$h[\![a!\langle b@k\rangle]\!] \mid k[\![b!\langle\rangle]\!] \quad \text{and} \quad h[\![a!\langle b@k\rangle]\!] \mid k[\![\mathsf{stop}]\!]$$

and Γ the environment

$$h : \mathsf{loc},\ h : \mathbf{move},\ k : \mathsf{loc},\ a : \mathsf{rw}\langle \mathsf{r}\langle\rangle@k\rangle@h,\ b : \mathsf{w}\langle\top\rangle@k$$

Then if k is in $\mathcal{T}$ one can check that $N_5 \not\approx^{\mathcal{T}}_{bis} N_6$. This is because $\Gamma \rhd N_5$ can perform the action $h.a!b@k$ followed by $k.b!\langle\rangle$, which can not be matched by $\Gamma \rhd N_6$.

However $\Gamma \models N_5 \cong^{\mathcal{T}}_{cxt} N_6$ because it is not possible to find a context to distinguish between them. A context can be found to augment the knowledge of the environment at h with the read capability for b at k. But, in a well-typed context, it is not possible to transfer this information from h to where it can be put to use, namely k.

This example demonstrates that even with our very restricted move capability there are problems with the flow of information. Knowledge about the system learnt at l can not necessarily be passed to k if the environment does not have move capability at k. This motivates the new form of configurations we introduce for the labelled transition system necessary in order to characterise $\cong^{\mathcal{T}}_{cxt}$.

We replace a simple Γ with a structure $\overline{\Gamma} = (\Gamma, \Gamma_{k_1}, \ldots, \Gamma_{k_n})$ where the k_i make up $\mathcal{T}$. Each Γ_{k_i} represents localised knowledge at k_i whereas Γ represents the centralised knowledge, available at any location for which we have move capability. Given that we can store the centralised knowledge at a location k_0, provided by the environment (with move capability), we can always pass local knowledge on to k_0 (but not vice versa). Thus centralised knowledge is always greater than any of the local knowledge environments. This leads us to the following definition:

Definition 6 (Configurations). *A* configuration *$(\overline{\Gamma} \rhd M)$ over $\mathcal{T}$ consists of a family of type environments $\overline{\Gamma} = \Gamma, \Gamma_{k_1}, \ldots, \Gamma_{k_n}$ such that*

Fig. 6. Labelled transition rules accounting for the move capability

$$\frac{\Gamma \vdash k : \mathsf{loc}[\mathbf{move}] \quad a : \mathsf{r}\langle \mathrm{T}\rangle_{@k} \in \Gamma \quad \Gamma \sqcap \langle v : \mathrm{T}\rangle_{@k} \text{ exists}}{(\overline{\Gamma} \rhd k[\![a!\langle v\rangle\, P]\!]) \xrightarrow{k.a!v} (\overline{\Gamma} \sqcap_0 \langle v : \mathrm{T}\rangle_{@k} \rhd k[\![P]\!])} \quad (\text{LTS-MOVE}-\text{OUT})$$

$$\frac{\Gamma \not\vdash k : \mathsf{loc}[\mathbf{move}] \quad k \in \mathcal{T} \quad a : \mathsf{r}\langle \mathrm{T}\rangle_{@k} \in \Gamma_k \quad \overline{\Gamma} \sqcap_0 \langle v : \mathrm{T}\rangle_{@k} \sqcap_k \langle v : \mathrm{T}\rangle_{@k} \text{ exists}}{(\overline{\Gamma} \rhd k[\![a!\langle v\rangle\, P]\!]) \xrightarrow{k.a!v} (\overline{\Gamma} \sqcap_0 \langle v : \mathrm{T}\rangle_{@k} \sqcap_k \langle v : \mathrm{T}\rangle_{@k} \rhd k[\![P]\!])} \quad (\text{LTS-T}-\text{OUT})$$

$$\frac{\Gamma \vdash k : \mathsf{loc}[\mathbf{move}] \quad \Gamma \vdash a : \mathsf{w}\langle \mathrm{T}\rangle_{@k} \quad \Gamma \vdash v : \mathrm{T}_{@k}}{(\overline{\Gamma} \rhd k[\![a?(X : \mathrm{A})\, P]\!]) \xrightarrow{k.a?v} (\overline{\Gamma} \rhd k[\![P\{\!\{v/X\}\!\}]\!])} \quad (\text{LTS-MOVE}-\text{IN})$$

$$\frac{\Gamma \not\vdash k : \mathsf{loc}[\mathbf{move}] \quad k \in \mathcal{T} \quad \Gamma_k \vdash a : \mathsf{w}\langle \mathrm{T}\rangle_{@k} \quad \Gamma_k \vdash v : \mathrm{T}_{@k}}{(\overline{\Gamma} \rhd k[\![a?(X : \mathrm{A})\, P]\!]) \xrightarrow{k.a?v} (\overline{\Gamma} \rhd k[\![P\{\!\{v/X\}\!\}]\!])} \quad (\text{LTS-T}-\text{IN})$$

$$\frac{(\overline{\Gamma}, \overline{(n : T)}_{\nabla} \rhd M) \xrightarrow{(\tilde{n}:\tilde{\mathrm{T}})k.a?V} (\overline{\Gamma}' \rhd M') \quad n \neq a, k}{(\overline{\Gamma} \rhd M) \xrightarrow{(n:\mathrm{T}\tilde{n}:\tilde{\mathrm{T}})k.a?V} (\overline{\Gamma}' \rhd M')} \quad (\text{LTS-T}-\text{WEAK})$$

- $\mathcal{T} = \{k_1, \ldots, k_n\}$
- $\Gamma <: \Gamma_{k_i}$ *for each* $1 \leq i \leq n$

and a system M *which can be typed in some* extension *of* Γ.

We will write $\overline{\Gamma}^{\mathcal{T}}_{\nabla}$ to mean the family of environments $\Gamma, \Gamma_{k_1}, \ldots, \Gamma_{k_n}$ such that each component Γ_{k_i} is equal to the environment Γ; we will typically omit the parameter $\mathcal{T}$ here as it can usually be recovered from context. We understand $\overline{\Gamma}, \overline{\Gamma}'$ and $\overline{\Gamma} \sqcap \overline{\Gamma}'$ to be pointwise operations. Finally we need a notation for increasing knowledge in the individual components of a configuration, for which we use the notation $\sqcap_k$. For instance we write $\overline{\Gamma} \sqcap_0 \Gamma'$ to mean the family

such that the global component becomes $\Gamma \sqcap_0 \Gamma'$ and all other components are unchanged. Similarly $\overline{\Gamma} \sqcap_k \Gamma'$ adds, if possible, Γ' to the k^{th} component.

We define our new labelled transition system, parameterised on $\mathcal{T}$, as binary relations between these new configurations. We replace the rules (LTS-OUT) and (LTS-IN) in Figure 5 with those in Figure 6 and modifying the remaining rules in Figure 5 in the obvious manner. The standard definition of (weak) bisimilarity may be applied to this new labelled transition system. To emphasise the role of the parameter $\mathcal{T}$ we will write the resulting equivalence as $\approx^{\mathcal{T}}_{bis}$.

We can now state the final result of the paper that, by considering a configuration in which the knowledge at every locality is initially Γ, then bisimilarity coincides with reduction barbed congruence with respect to Γ:

Theorem 3 (Full abstraction). *In* DPI *with restricted mobility*

$$\Gamma \models M \mathrel{\overline{\approx}}^{\mathcal{T}}_{cxt} N \qquad \textit{iff} \qquad \overline{\Gamma}_{\triangledown} \models M \approx^{\mathcal{T}}_{bis} N.$$

6 Conclusions and Related Work

We have presented two labelled transition systems for which bisimilarity coincides with a natural notion of contextual equivalence for distributed systems. The labelled transitions rely upon a type discipline for the language which can control resource access and mobility. As in [8,2], the use of a type environment representing the tester's knowledge of the system plays an important role in characterising the contextual equivalences. In particular it aided us in defining a labelled transition system which accounts for information flow in a distributed setting with restricted mobility.

There has been a great deal of interest in modelling distributed systems using calculi in recent years, [14,6,1,4,16,9,3]. The emphasis so far has largely been on design of the languages to give succinct descriptions of mobile processes with type systems given to constrain behaviour in a safe manner. Where equivalence has been used it has typically been introduced as some sort of contextual equivalence very similar to the one found in the present paper [6,1,11]. Proofs of correctness of protocols or language translations have been carried out with respect to these contextual equivalences. Recently in [5] a form of bisimulation has been suggested as a proof method for establishing contextual equivalence in the Seal calculus; but, as far as we know, the only existing example of an operational characterisation of behavioural equivalence in the distributed setting is found in [10].

The work in [15] takes a different, more intensional approach to equivalence in the distributed setting in that, in order to establish correctness of a particular protocol, a novel notion of equivalence based on coupled simulation tailored to accommodate migration is identified. Although having many interesting properties such as congruence, this equivalence is not shown to coincide with any independent contextually defined notion of equivalence.

References

1. Roberto M. Amadio and Sanjiva Prasad. Modelling IP mobility. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR '98: Concurrency Theory (9th International Conference, Nice, France)*, volume 1466 of *LNCS*, pages 301–316. Springer, September 1998.
2. M. Boreale and D. Sangiorgi. Bisimulation in name-passing calculi without matching. In *13th LICS Conf.* IEEE Computer Society Press, 1998.
3. Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995. Short version in *Proceedings of POPL '95*. A preliminary version appeared as Report 122, Digital Systems Research, June 1994.
4. Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, June 2000.
5. G. Castagna and F. Zappa. The seal calculus revisited. In *22th Conference on the Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, 2002.
6. Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, August 26-29 1996. Springer-Verlag. LNCS 1119.
7. M. Hennessy, M. Merro, and J. Rathke. Towards a behavioural theory of access and mobility control in distributed systems. Computer Science Report 2002:01, University of Sussex, 2002.
8. M. Hennessy and J. Rathke. Typed behavioural equivalences for processes in the presence of subtyping. In *Proc. CATS2002, Computing: Australasian Theory Symposium, Melbourne 2002*, 2002. Also available as a University of Sussex technical report.
9. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.
10. M. Merro and M. Hennessy. Bisimulation congruences in safe ambients. *ACM SIGPLAN Notices*, 31(1):71–80, January 2002.
11. M. Merro, J. Kleist, and U. Nestmann. Mobile Objects as Mobile Processes. *To appear in Journal of Information and Computation*, 2002.
12. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
13. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.
14. Peter Sewell. Global/local subtyping and capability inference for a distributed pi-calculus. In *ICALP 98*, volume 1443 of *LNCS*. Springer, 1998.
15. Asis Unyapoth and Peter Sewell. Nomadic pict: Correct communication infrastructure for mobile computation. In *Conference Record of POPL'01: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 236–247, London, United Kingdom, January 17–19, 2001.
16. J. Vitek and G. Castagna. A calculus of secure mobile computations. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *LNCS*. Springer, 1999.

The Two-Variable Guarded Fragment with Transitive Guards Is 2EXPTIME-Hard*

Emanuel Kieroński

Institute of Computer Science
University of Wrocław
ul. Przesmyckiego 20, 51-151 Wrocław, Poland
kiero@ii.uni.wroc.pl

Abstract. We prove that the satisfiability problem for the two-variable guarded fragment with transitive guards $\mathsf{GF}^2 + TG$ is 2EXPTIME-hard. This result closes the open questions left in [4], [17]. In fact, we show 2EXPTIME-hardness of $min\mathsf{GF}^2 + TG$, a fragment of $\mathsf{GF}^2 + TG$ without equality and with just one transitive relation $\prec$, which is the only non-unary symbol allowed. Our lower bound for $min\mathsf{GF}^2 + TG$ matches the upper bound for the whole guarded fragment with transitive guards and the unrestricted number of variables $\mathsf{GF} + TG$ given by Szwast and Tendera [17], so in fact $\mathsf{GF}^2 + TG$ is 2EXPTIME-complete. It is surprising that the complexity of $min\mathsf{GF}^2 + TG$ is higher then the complexity of the variant with *one-way* transitive guards $\mathsf{GF}^2 + \overrightarrow{TG}$ [9]. The latter logic appears naturally as a counterpart of temporal logics without past operators.

1 Introduction

1.1 Modal Logic versus First-Order Logic

The first order logic[1] is a very natural and convenient language for expressing properties of many systems that can be encountered in various areas of computer science. Unfortunately, this convenience and expressive power are expensive and cause that decision problems for the first order logic are difficult to solve algorithmically. In particular it has been known since works of Church and Turing in 1930's, that the satisfiability problem for the first order logic is undecidable.

On the other hand *(propositional) modal and temporal logics* are widely used in computer science. Their satisfiability problems are decidable and they posses a lot of other good algorithmic and model-theoretic properties. Therefore, they have a lot of applications in database theory, artificial intelligence, verification of hardware and software, etc.

Propositional modal logic can be translated into the first order logic. The image of this translation, the so-called *modal fragment*, is a very restricted fragment

* Supported by KBN grant 8 T11C 043 19

[1] By the first order logic we mean in this paper the first order logic without constants and function symbols.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 299–312, 2003.

of the first-order logic. Researchers in computer science would like to extend the modal fragment to obtain such fragments of the first order logic which are still decidable and retain good properties of modal logic but are more powerful. Finding such an extension could also provide an explanation of good properties of modal logics.

Since the modal fragment is a two-variable logic, the two-variable first-order logic FO^2 was considered a good candidate for such an extension. The decidability of the satisfiability problem for FO^2 was proved by Mortimer [14]. Later it turned out that it is NEXPTIME-complete. The lower bound was given by Lewis [12] and the upper bound by Grädel, Kolaitis and Vardi [7], who established the exponential model property. Unfortunately, though decidable, FO^2 lacks some good properties of modal logic. For example, if we extend FO^2 by fixed point operators, we obtain a logic which is undecidable [6] in contrast to the μ-calculus [10] – the propositional modal logic augmented with fixed point operators. A lot of algorithmic problems is also caused by the fact that FO^2 does not posses a tree model property.

Another extension of the modal fragment, the so called *guarded fragment*, was proposed by Andréka, van Benthem and Németi [1].

1.2 Guarded Fragment of First Order Logic

In the guarded fragment GF we do not restrict the number of variables or the arity of relation symbols. Some restrictions are imposed on the usage of quantifiers but we do not demand a special prefix of quantifiers, which is usual in many known decidable fragments of the first-order logic. In GF, every quantifier has to be relativized by an atomic formula containing all variables that are free in the scope of this quantifier.

Definition 1. (Andréka, van Benthem and Németi [1]) *The* guarded fragment GF *of the first order logic is defined inductively:*

1. *Every atomic formula belongs to* GF.
2. GF *is closed under* $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.
3. *If* $\boldsymbol{x}, \boldsymbol{y}$ *are tuples of variables,* $\alpha(\boldsymbol{x}, \boldsymbol{y})$ *is an atomic formula,* $\psi(\boldsymbol{x}, \boldsymbol{y})$ *is in* GF *and* $\mathit{free}(\psi) \subseteq \mathit{free}(\alpha) = \{\boldsymbol{x}, \boldsymbol{y}\}$*, where* $\mathit{free}(\phi)$ *is the set of free variables of* ϕ*, then formulas*

$$\exists \boldsymbol{y}(\alpha(\boldsymbol{x}, \boldsymbol{y}) \wedge \psi(\boldsymbol{x}, \boldsymbol{y})),$$

$$\forall \boldsymbol{y}(\alpha(\boldsymbol{x}, \boldsymbol{y}) \rightarrow \psi(\boldsymbol{x}, \boldsymbol{y}))$$

belong to GF.

Atoms $\alpha(\boldsymbol{x}, \boldsymbol{y})$, that relativize quantifiers in the above definition, are called *guards*. The order of variables in a guard $\alpha(\boldsymbol{x}, \mathbf{y})$ can be arbitrary.

Let us consider some examples. It is easy to express in GF that a binary relation R is symmetric:

$$\forall x, y\ (R(x, y) \rightarrow R(y, x)).$$

Unfortunately, a formula stating that a binary relation R is transitive:

$$\forall x, y, z\ ((R(x, y) \wedge R(y, z)) \rightarrow R(x, z))$$

is not in GF, since the formula $R(x, y) \wedge R(y, z)$, relativizing the quantifier, is not atomic.

It has been shown that GF retains a lot of good properties of modal and temporal logic. In particular, Grädel proved that it has the finite model property, and that its every satisfiable formula has a tree-like model [5]. The satisfiability problem is decidable and has double exponential complexity. The reason for such a high complexity is the unrestricted number of variables. In practical applications only several variables are usually used. The bounded version GF^k of GF, allowing only k variables, is EXPTIME-complete. Grädel and Walukiewicz investigated *guarded fixed point logic* $\mu\mathsf{GF}$ [8]. They proved that adding least and greatest fixed point operators to GF gives a decidable logic, and moreover, does not increase the complexity, i.e. the satisfiability problem for $\mu\mathsf{GF}$ is in 2EXPTIME and for $\mu\mathsf{GF}^k$ – in EXPTIME. It is worth mentioned, that $\mu\mathsf{GF}$ is a very powerful logic. For example, even its two-variable version without equality allows to encode μ-calculus with backward modalities.

1.3 Guarded Fragment with Transitive Relations

In some modal logics transitivity axioms are built-in. Such logics would be conveniently embedded in the guarded fragment if we could specify that some binary relations are transitive. As we have seen, a straightforward idea of expressing transitivity of a binary relation leads to a formula which is not properly guarded. In fact, it can be shown that transitivity cannot be expressed in GF in any other way.

Grädel [5] considered an extension of GF: we require that some binary relations are transitive. He proved that GF^3 with transitivity statements is undecidable. This result was then improved by Ganzinger, Meyer and Veanes [4] who proved that even GF^2 without equality and with transitivity is undecidable. On the other hand they observed that the two-variable, *monadic* guarded fragment with transitivity $\mathsf{MGF}^2 + TG$ is decidable. A guarded formula is monadic if all of its non-unary predicates can appear only in guards. It seems that $\mathsf{MGF}^2 + TG$ is very close to modal logic with transitivity axioms but as pointed out by Ganzinger, Meyer and Veanes it is stronger since it does not have the finite model property.

1.4 Guarded Fragment with Transitive Guards

In the acronym $\mathsf{MGF}^2 + TG$ letters TG denote *transitive guards*. Indeed, if a formula is monadic then in particular all binary and hence all transitive symbols can appear only in guards.

Ganzinger, Meyer and Veanes did not give good complexity estimates for $\mathsf{MGF}^2 + TG$ since their proof was obtained by a reduction to Rabin's theory of k

successors [16]. They also left another, natural open question – the decidability of $\mathsf{GF}{+}TG$, the whole guarded fragment with transitive relations, where transitive relations are admitted only in guards but where non-transitive relations and equality can occur elsewhere. The last question was answered by Szwast and Tendera [17] who proved that $\mathsf{GF}{+}TG$ is decidable, and that its satisfiability problem is in 2EXPTIME.

Surprisingly, the two-variable guarded fragment with transitive guards is not in EXPTIME. By an elegant reduction from FO^2 Szwast and Tendera showed that it is NEXPTIME-hard. In [9] we slightly improved this bound. We introduced the two-variable guarded fragment with *one-way* transitive guards $\mathsf{GF}^2 + \overrightarrow{TG}$, which is a proper subset of $\mathsf{GF}^2 + TG$, and showed that its satisfiability problem is EXPSPACE-complete. In this paper we close the remaining gap and prove that the satisfiability problem for $\mathsf{GF}^2 + TG$ is 2EXPTIME-hard.

Our 2EXPTIME lower bound is obtained for a very restricted version of $\mathsf{GF}^2 + TG$ denoted by $min\mathsf{GF}^2 + TG$. This logic does not allow equality and contains only one transitive relation $\prec$, which is the only non-unary symbol. The lack of equality reduces the expressive power of the logic since it is impossible to define cliques. With equality we can enforce that every model of a formula contains transitive cliques of size exponential with respect to the size of the formula.

We believe that this lower bound is surprising for at least two reasons. First, it matches the upper bound for the whole $\mathsf{GF}{+}TG$. It is not usual that the complexity of the two-variable version of a logic equals the complexity of the same logic with the unbounded number of variables. The second reason concerns the correspondence between the guarded fragment with transitive guards and branching temporal logics and will be explained in the next subsection.

1.5 Guarded Fragment with One-Way Transitive Guards

What do we mean by the guarded fragment with one-way transitive guards $\mathsf{GF}{+}\overrightarrow{TG}$? Consider a subformula of the form $\exists y\, \alpha(x,y) \wedge \psi(x,y)$, where a binary, transitive symbol $\prec$ is used in the guard α. There are two possibilities $\alpha(x,y) = x \prec y$ or $\alpha(x,y) = y \prec x$. In $\mathsf{GF}{+}\overrightarrow{TG}$ we allow only the first one[2]. This is similar to the situation in most temporal logics where we can quantify points in the future but not in the past. In this context, our results become quite interesting: we expose the difference in the complexity of reasoning about the future only and both the future and the past, in the two-variable guarded fragment with transitive guards.

If we interpret the only binary symbol $\prec$ in $min\mathsf{GF}^2 + TG$ as a relation representing time then we can consider $min\mathsf{GF}^2 + TG$ a counterpart of a simple branching temporal logic with both future and past operators. This logic is 2EXPTIME-hard. On the other hand when we disallow past then, even if we allow equality and additional, transitive and non-transitive, binary relations

[2] Our choice of the first possibility is not essential and similar results can be obtained when the second possibility is chosen.

("modalities"), we get a logic which is EXPSPACE-complete, so has lower complexity (assuming hypothesis EXPSPACE $\neq$2EXPTIME).

This is rather surprising since for temporal and process logics the complexities of the satisfiability problem for versions with past operators and versions without them usually coincide. For example Kupferman and Pnueli [11] investigated two variants of CTL augmented with past operators. Both of them turned out to be EXPTIME-complete (similarly to CTL). Adding inverse modalities to μ-calculus does not increase its complexity [18]. Also propositional dynamic logic PDL and its version with converse CPDL have the same complexity [13,15].

2 Preliminaries

In Definition 1 we introduced the guarded fragment GF of the first-order logic. Then we informally introduced some of its variants and extensions. Let us give the formal definition of the variants we are interested in this paper.

Definition 2. Guarded fragment with transitive guards GF+TG *is an extension of* GF *by transitive relations. A* GF+TG *formula* Φ *consists of a* GF *formula* Φ' *and a list* $\mathcal{T}$ *of binary relation symbols that are required to be transitive. A relation symbol* T *can belong to* $\mathcal{T}$ *if it appears in* Φ *only in guards. We say that* Φ *is satisfiable if* Φ' *is satisfiable in a model in which all interpretations of symbols from* $\mathcal{T}$ *are transitive.*

We distinguish a restricted, *minimal* version of the considered logic, for which we prove our lower bound:

Definition 3. *By* minGF$^2 + TG$ *we denote the fragment of the two-variable* GF$+TG$ *without equality, which contains only one non-unary symbol* $\prec$*. Symbol* $\prec$ *is binary and can be used only in guards.*

An *alternating Turing machine* is a generalization of a nondeterministic Turing machine. States, and hence configurations of such a machine, are partitioned into four groups: existential, universal, accepting and rejecting. The notion of an accepting (rejecting) configuration is extended to the case of configurations in which states are existential or universal. This can be done inductively: a universal configuration is accepting if all its successor configurations, i.e. configurations obtained by performing one machine step according to the transition function, are accepting and an existential configuration is accepting if at least one of its successor configurations is accepting. A machine M accepts its input w if the initial configuration of M on w is accepting. By ASPACE(f) we denote the set of problems that can be solved by alternating Turing machines working in space bounded by a function f. In this paper we will use the following theorem:

Theorem 1. (Chandra, Kozen, Stockmeyer [3]) *For* $t(n) \geq \log n$*:*

$$ASPACE[t(n)] = DTIME[2^{t(n)}].$$

Let $AEXPSPACE = \bigcup_{k=1}^{\infty} ASPACE(2^{n^k})$. *Then in particular:*

$$AEXPSPACE = 2EXPTIME.$$

For details about alternating Turing machines you can see for example [2].

3 Proof

In this section we prove the main result of the paper.

Theorem 2. *The satisfiability problem for* $min\mathsf{GF}^2 + TG$ *is 2EXPTIME-hard.*

Proof. By Theorem 1, it suffices to prove that every problem in AEXPSPACE can be reduced in polynomial time to the satisfiability problem for $min\mathsf{GF}^2+TG$.

Let M be an alternating Turing machine working in space bounded by 2^{n^k}. Let w be an input for M. We construct a $min\mathsf{GF}^2 + TG$ sentence Φ which is satisfiable if and only if M accepts w. Without any loss of generality we can assume that in every configuration, M has exactly two possible transitions and that on its every computation path it enters an accepting or rejecting state at exactly $2^{2^{n^k}}$-th step. To simplify our proof we assume that after entering an accepting or rejecting state, M does not stop. More precisely, we assume that accepting and rejecting states are universal. In each of such states, M has two identical transitions: it does not write any symbol on the tape and it does not move its head. In other words, after accepting or rejecting M stays infinitely in the same configuration.

Every configuration will be represented by a set of 2^{n^k} elements, each of them corresponding to a single cell of the tape. To encode the position of an element in a configuration, i.e. the consecutive number of the tape cell it represents, we use unary relation symbols $P_1, \ldots, P_{n^k}$. Formally, $P_i(a)$ is true if the i-th bit of the position of the element a is set to 1. We use the abbreviation $\overline{P}(a)$ to describe this position ($0 \leq \overline{P}(a) < 2^{n^k}$).

Observe that it is not difficult to express the following properties with formulas of polynomial length:

$$\overline{P}(x) = l \text{ for fixed } l,\ 0 \leq l < 2^{n^k},$$
$$\overline{P}(x) \geq l \text{ for fixed } l,\ 0 \leq l < 2^{n^k},$$
$$\overline{P}(x) = \overline{P}(y),$$
$$\overline{P}(x) = \overline{P}(y) + 1.$$

For example the last property can be expressed as follows:

$$\bigvee_{0 \leq i < n^k} \big(P_i(x) \wedge \neg P_i(y) \wedge \bigwedge_{j>i} (\neg P_j(x) \wedge P_j(y)) \wedge \bigwedge_{j<i} (P_j(x) \leftrightarrow P_j(y))\big).$$

We connect each pair of elements a, b, such that $\overline{P}(a) < \overline{P}(b)$, belonging to the same configuration, with the transitive symbol $\prec$, i.e. we want $a \prec b$ to be true in our model. Figure 1 gives a representation of a configuration.

Fig. 1. A representation of an example configuration

We describe a configuration in a standard way: for each symbol a_i in the alphabet of M (including **blank**) we use a unary relation symbol A_i, for each state q_i – a unary symbol Q_i. We also have a unary symbol H describing the head position. For element x representing a tape cell scanned by the head, $H(x)$ and $Q_i(x)$, for some i, will be true.

We begin our construction by enforcing that every model of Φ contains a substructure that can be viewed as an infinite binary tree. The set of 2^{n^k} elements describing a single configuration of M will be treated as a "node" of this tree. In fact, we are not able to enforce that a model contains a "real" tree. For example, it is possible that some of elements of a model represent more than one tape cell. What we do is ensure that every node can identify its two successor nodes.

We organize the structure in such a way that elements belonging to an even configuration, i.e. a configuration whose depth in a computation tree is even, will be smaller, with respect to relation $\prec$, than elements belonging to its successor configurations, and elements belonging to an odd configuration will be greater than elements belonging to its successor configurations. We do not impose any relations between elements that do not belong the same configuration or to two consecutive configurations. Additionally, we introduce unary symbols D_0, D_1, D_2, D_3 and enforce that D_i is true exactly for elements belonging to configurations whose number is of the form $4k + i$. One more unary symbol L will indicate that the element belongs to the left son of some node.

The structure of the tree is shown in Fig. 2. Horizontal arrows represent configurations (that are internally ordered by $\prec$ as described above). Orientation of arrows represents the relation $\prec$.

First, we express that for every element of a model at most one of unary relations D_i is true:

$$\Phi_1 \equiv \bigwedge_i (\forall x\ (D_i(x) \rightarrow \bigwedge_{j \neq i} \neg D_j(x))).$$

Formulas Φ_2 and Φ_3 say that there exists a node representing the initial configuration of M on w. For all elements of this node, the special unary symbol I is true. We assume that this configuration is a left configuration.

$$\Phi_2 \equiv \exists x\ (I(x) \wedge D_0(x) \wedge L(x) \wedge \overline{P}(x) = 2^{n^k} - 1),$$

$$\begin{aligned}\Phi_3 \equiv \forall x\ (I(x) \rightarrow (\overline{P}(x) \neq 0 \\ \rightarrow\ \ \exists y\ (y \prec x \wedge I(y) \wedge D_0(y) \wedge L(y) \wedge \overline{P}(x) = \overline{P}(y) + 1))).\end{aligned}$$

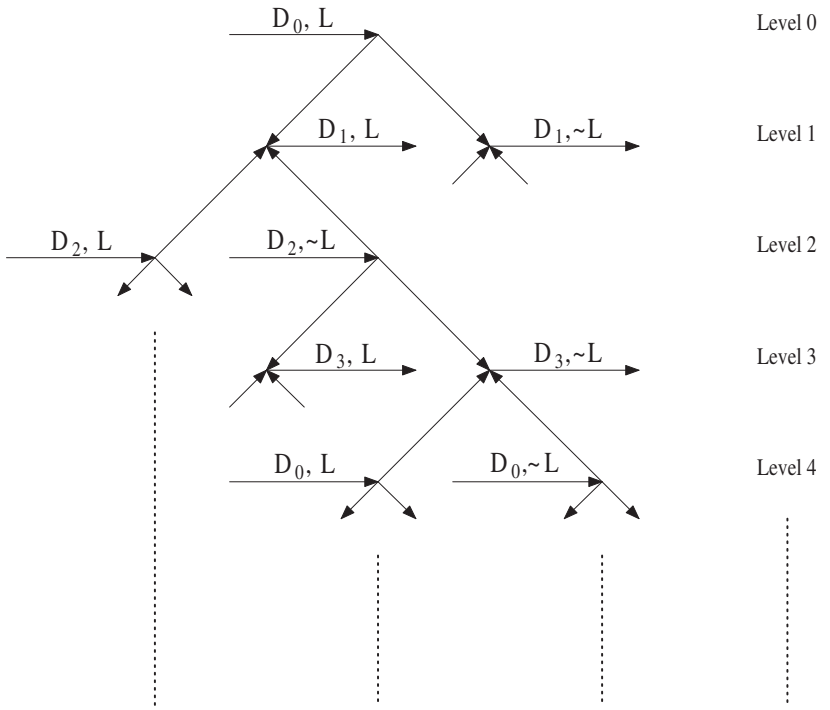


Fig. 2. The structure of the tree

Formulas $\Phi_4 - \Phi_7$ express that for every element, except the first one, belonging to a description of an even configuration there exists a predecessor in this configuration, and for every element, except the last one, belonging to a description on an odd configuration there exists a successor in this configuration:

$$\begin{aligned}
\Phi_4 \equiv \forall x \ \ & (D_0(x) \rightarrow (\overline{P}(x) \neq 0 \\
& \rightarrow \quad \exists y \ (y \prec x \wedge D_0(x) \wedge (L(x) \leftrightarrow L(y)) \wedge \overline{P}(x) = \overline{P}(y) + 1))), \\
\Phi_5 \equiv \forall x \ \ & (D_1(x) \rightarrow (\overline{P}(x) \neq 2^{n^k} - 1 \\
& \rightarrow \quad \exists y \ (x \prec y \wedge D_1(x) \wedge (L(x) \leftrightarrow L(y)) \wedge \overline{P}(y) = \overline{P}(x) + 1))), \\
\Phi_6 \equiv \forall x \ \ & (D_2(x) \rightarrow (\overline{P}(x) \neq 0 \\
& \rightarrow \quad \exists y \ (y \prec x \wedge D_2(x) \wedge (L(x) \leftrightarrow L(y)) \wedge \overline{P}(x) = \overline{P}(y) + 1))), \\
\Phi_7 \equiv \forall x \ \ & (D_3(x) \rightarrow (\overline{P}(x) \neq 2^{n^k} - 1 \\
& \rightarrow \quad \exists y \ (x \prec y \wedge D_3(x) \wedge (L(x) \leftrightarrow L(y)) \wedge \overline{P}(y) = \overline{P}(x) + 1))).
\end{aligned}$$

For every node of the tree there exist left and right successor nodes. For a node representing an even configuration, we connect successors directly to the element a which is the last element in this configuration and enforce that a is smaller than

elements in successors. For a node representing an odd configuration, successors are connected to the first element a in the configuration and the element a is made greater than its successors. The existence of appropriate successors will be implied by formulas $\Phi_8 - \Phi_{11}$.

$$\begin{aligned}
\Phi_8 \equiv \forall x\, \big(D_0(x) \to (\overline{P}(x) = 2^{n^k} - 1 \to (\exists y\, (x \prec y \wedge D_1(y) \wedge L(y) \wedge \overline{P}(y) = 0) \\
\wedge \exists y\, (x \prec y \wedge D_1(y) \wedge \neg L(y) \wedge \overline{P}(y) = 0)))\big), \\
\Phi_9 \equiv \forall x\, \big(D_1(x) \to (\overline{P}(x) = 0 \to (\exists y\, (y \prec x \wedge D_2(y) \wedge L(y) \wedge \overline{P}(y) = 2^{n^k} - 1) \\
\wedge \exists y\, (y \prec x \wedge D_2(y) \wedge \neg L(y) \wedge \overline{P}(y) = 2^{n^k} - 1)))\big), \\
\Phi_{10} \equiv \forall x\, \big(D_2(x) \to (\overline{P}(x) = 2^{n^k} - 1 \to (\exists y\, (x \prec y \wedge D_3(y) \wedge L(y) \wedge \overline{P}(y) = 0) \\
\wedge \exists y\, (x \prec y \wedge D_3(y) \wedge \neg L(y) \wedge \overline{P}(y) = 0)))\big), \\
\Phi_{11} \equiv \forall x\, \big(D_3(x) \to (\overline{P}(x) = 0 \to (\exists y\, (y \prec x \wedge D_0(y) \wedge L(y) \wedge \overline{P}(y) = 2^{n^k} - 1) \\
\wedge \exists y\, (y \prec x \wedge D_0(y) \wedge \neg L(y) \wedge \overline{P}(y) = 2^{n^k} - 1)))\big),
\end{aligned}$$

We introduce two abbreviations which will help to present some of the remaining formulas in a more succinct way. The formula $SameLetter(x, y)$ says that elements x and y are marked with the same unary symbol from the set $\{D_0, D_1, D_2, D_3\}$. $NEXT(x, y)$ is true for two consecutive elements of a description of a configuration.

$$\begin{aligned}
SameLetter(x, y) &\equiv (D_0(x) \leftrightarrow D_0(y)) \wedge (D_1(x) \leftrightarrow D_1(y)) \\
&\quad \wedge (D_2(x) \leftrightarrow D_2(y)) \wedge (D_3(x) \leftrightarrow D_3(y)), \\
Next(x, y) &\equiv x \prec y \wedge SameLetter(x, y) \wedge \overline{P}(y) = \overline{P}(x) + 1.
\end{aligned}$$

Now we say that a model of our formula satisfies several basic properties of a computation tree. Φ_{12} states that there is exactly one alphabet symbol in every tape cell.

$$\Phi_{12} \equiv \bigwedge_j \big(\forall x\, (D_j(x) \to \bigvee_i A_i(x))\big) \quad \wedge \quad \bigwedge_j \big(\forall x\, (A_j(x) \to \bigwedge_{i \neq j} \neg A_i(x))\big).$$

Formulas $\Phi_{13} - \Phi_{14}$ say that in each configuration at most one element is scanned by the head.

$$\begin{aligned}
\Phi_{13} \equiv \forall x\, \big(H(x) \to \forall y\, (x \prec y \to (SameLetter(x, y) \to \neg H(y)))\big), \\
\Phi_{14} \equiv \forall x\, \big(H(x) \to \forall y\, (y \prec x \to (SameLetter(x, y) \to \neg H(y)))\big).
\end{aligned}$$

Φ_{15} says that exactly those elements which represent tape cells observed by the head store information about state.

$$\Phi_{15} \equiv \bigwedge_i \big(\forall x\, (Q_i(x) \to H(x))\big) \quad \wedge \quad \forall x\, \big(H(x) \to \bigvee_i Q_i(x)\big).$$

Formulas $\Phi_{16} - \Phi_{18}$ ensure that the root of the tree describes the initial configuration of M, in the initial state q_0, on the input $w = w_0 \dots w_{n-1}$.

$$\Phi_{16} \equiv \forall x \, \big(I(x) \to (\overline{P}(x) = 0 \to (H(x) \wedge Q_0(x)))\big),$$
$$\Phi_{17} \equiv \bigwedge_{i<n} \forall x \, (I(x) \to (\overline{P}(x) = i \to W_i(x))),$$
$$\Phi_{18} \equiv \forall x \, \big(I(x) \to (\overline{P}(x) \geq n \to BLANK(x))\big).$$

Formulas $\Phi_{19} - \Phi_{20}$ express that if a tape cell of a configuration is not scanned by the head then in the same cell of both successor configurations the alphabet symbol does not change.

$$\Phi_{19} \equiv \forall x, y \, \big(x \prec y \to \big(((D_0(x) \wedge D_1(y) \vee D_2(x) \wedge D_3(y)) \wedge \neg H(x) \wedge \overline{P}(x) = \overline{P}(y)\big) \to \bigwedge_i (A_i(x) \leftrightarrow A_i(y)\big),$$
$$\Phi_{20} \equiv \forall x, y \, \big(y \prec x \to \big(((D_1(x) \wedge D_2(y) \vee D_3(x) \wedge D_0(y)) \wedge \neg H(x) \wedge \overline{P}(x) = \overline{P}(y)\big) \to \bigwedge_i (A_i(x) \leftrightarrow A_i(y)\big).$$

Consider now a node t of a tree and a configuration c that is described by this node. There are two cases: the state of the machine in this configuration is existential or it is universal.

In the first case we enforce that the configuration represented by the left son of t is created by applying one of the two possible transitions on c. Assume that for an existential state q and a letter a there are two possible transitions: $(q, a) \to (q', a', \to)$ and $(q, a) \to (q'', a'', \leftarrow)$.

We put:

$$\begin{aligned}\Phi^{exists}_{(a,q)} \equiv \forall x, y \, \big(x \prec y \to \big(((D_0(x) \wedge D_1(y) \vee D_2(x) \wedge D_3(y)) \\ \wedge Q(x) \wedge A(x) \wedge L(y) \wedge \overline{P}(x) = \overline{P}(y)) \\ \to \big((A'(y) \wedge \forall x \, (y \prec x \to (Next(y, x) \to H(x) \wedge Q'(x)))) \\ \vee (A''(y) \wedge \forall x \, (x \prec y \to (Next(x, y) \to H(x) \wedge Q''(x)))))\big)\big),\end{aligned}$$

$$\begin{aligned}\Phi^{exists'}_{(a,q)} \equiv \forall x, y \, \big(y \prec x \to \big(((D_1(x) \wedge D_2(y) \vee D_3(x) \wedge D_0(y)) \\ \wedge Q(x) \wedge A(x) \wedge L(y) \wedge \overline{P}(x) = \overline{P}(y)) \\ \to \big((A'(y) \wedge \forall x \, (y \prec x \to (Next(y, x) \to H(x) \wedge Q'(x)))) \\ \vee (A''(y) \wedge \forall x \, (x \prec y \to (Next(x, y) \to H(x) \wedge Q''(x)))))\big)\big).\end{aligned}$$

Other possible situations, when both transitions move the head forward, both transitions move the head backward, one of transitions does not move the head, etc., can be handled similarly.

Consider now the case of a universal configuration. We enforce that the left son of t is created by applying the first transition and the right son – the second

one. For a universal state q, a letter a and transitions $(q,a) \to (q',a',\rightarrow)$ and $(q,a) \to (q'',a'',\leftarrow)$ we put:

$$\begin{aligned}
\Phi^{univ}_{(a,q)} \equiv\ & \forall x,y\ \big(x \prec y \to \big(((D_0(x) \wedge D_1(y) \vee D_2(x) \wedge D_3(y)) \\
& \qquad \wedge Q(x) \wedge A(x) \wedge \overline{P}(x) = \overline{P}(y)) \\
& \to \big(\neg L(y) \to \big(A''(x) \wedge \forall x\ (x \prec y \to (Next(x,y) \to (Q''(x) \wedge H(x))))\big)\big) \\
& \wedge\ L(y) \to \big(A'(x) \wedge \forall x\ (y \prec x \to (Next(y,x) \to (Q'(x) \wedge H(x))))\big)\big)\big), \\
\Phi^{univ'}_{(a,q)} \equiv\ & \forall x,y\ \big(y \prec x \to \big(((D_1(x) \wedge D_2(y) \vee D_3(x) \wedge D_0(y)) \\
& \qquad \wedge Q(x) \wedge A(x) \wedge \overline{P}(x) = \overline{P}(y)) \\
& \to \big(\neg L(y) \to \big(A''(x) \wedge \forall x\ (x \prec y \to (Next(x,y) \to (Q''(x) \wedge H(x))))\big)\big) \\
& \wedge\ L(y) \to \big(A'(x) \wedge \forall x\ (y \prec x \to (Next(y,x) \to (Q'(x) \wedge H(x))))\big)\big)\big).
\end{aligned}$$

Note that if we had used only two relation symbols D_0, D_1 instead of four D_0, D_1, D_2, D_3 it would have caused problems with distinguishing between successors and predecessors of a configuration.

To finish our construction we give formula Φ_{21} stating that in none of configurations in a model, M is in its only rejecting state q_r.

$$\Phi_{23} \equiv \forall x\ (Q_r(x) \to \mathbf{false}).$$

We define Φ as

$$\Phi \equiv \bigwedge_{1 \le i \le 21} \Phi_i \wedge \bigwedge_{(a,q')} \Phi^{exist}_{a,q'} \wedge \bigwedge_{(a,q')} \Phi^{exist'}_{a,q'} \wedge \bigwedge_{(a,q'')} \Phi^{univ}_{a,q''} \wedge \bigwedge_{(a,q'')} \Phi^{univ'}_{a,q''},$$

where q' represents existential states and q'' represents universal states. Observe that the number of conjuncts, and the size of each of them are polynomial in $|M|$ and $|w|$.

We claim that Φ is satisfiable iff M accepts w. Indeed, if M accepts w then an accepting computation tree can be transformed into a model $\mathcal{M}$ of Φ in the following way. The root of the computation tree is transformed into the root of $\mathcal{M}$. Then we proceed recursively. Let c be a configuration in the computation tree and let c' be its code in $\mathcal{M}$. If c is universal then we transform its left subtree into the left subtree of c' and its right subtree into the right subtree of c'. If c is existential then we transform its accepting subtree into the left subtree of c'. Since we want to have a complete binary tree, we have to define somehow also the right subtree of c'. We can for example construct all nodes of this subtree in such a way that they agree with c' in predicates denoting alphabet symbols and for each element a from these nodes

$$\mathcal{M} \models \neg H(a) \wedge \bigwedge_i \neg Q_i(a).$$

It is easy to see that $\mathcal{M}$ is indeed a model of our formula.

For the proof of the opposite direction, we want to check that the existence of an accepting computation tree is implied by the existence of a model of Φ. A minor difficulty lies in the fact that this model is not necessarily a "real" tree, i.e. some elements of the model may represent more than one tape cell. Of course tape cells represented by the same element have the same position in the a configuration but can belong to different configurations. Nevertheless, we can simply obtain a tree model from an arbitrary model of Φ by taking the appropriate number of copies of such elements. Such a model corresponds to an accepting computation tree of M on w, with the exception that only left successors of existential nodes are correct. Now, for formal conformity, we transform the model into a computation tree by substituting right subtrees of existential nodes with subtrees which agree with transition function of M. This is not essential since in existential nodes we demand only one accepting successor and we know that the left one is in fact accepting. □

4 A Comment on the Proof

In the preliminary version of the proof some care was taken to provide a kind of the numbering of configurations. We used a unary symbol B. For an element a such that $\overline{P}(a) = l$, $B(a)$ was true in the model if and only if a belonged to a description of a configuration whose depth in a computation tree had the l-th bit set to 1. With such a numbering, we could encode computations of a usual alternating Turing machine which stops after accepting or rejecting. As pointed out to me by Jerzy Marcinkowski, when we enforce the machine to work infinitely we can get rid of the numbering of configurations.

5 Conclusion

The lower bound we gave in Theorem 2 and the upper bound given by Szwast and Tendera [17] lead to the following corollary:

Corollary 1. *The satisfiability problem for the two-variable guarded fragment with transitive guards* $\mathsf{GF}^2 + TG$ *is 2EXPTIME-complete.*

Since our result is obtained for $min\mathsf{GF}^2 + TG$, which is a subset of $\mathsf{MGF}^2 + TG$ we have also established the exact complexity bounds for $\mathsf{MGF}^2 + TG$.

In the table below we summarize the known results on the complexity of decidable extensions of the guarded fragment of first order logic. The result of this paper is denoted by *. The complexity of $\mathsf{GF} + \overrightarrow{TG}$ is implied by results for GF and $\mathsf{GF} + TG$.

References

1. H. Andréka, J. van Benthem, I. Németi. Modal Languages and Bounded Fragments of Predicate Logic. *Journal of Philosophical Logic*, 27, pages 217-274, 1998.

Table 1. Complexity of decidable extensions of GF

Complexity of decidable extensions of GF	
GF – 2EXPTIME [5]	GF^2 – EXPTIME [5]
μGF – 2EXPTIME [8]	$\mu\mathsf{GF}^2$ – EXPTIME [8]
$\mathsf{GF} + TG$ – 2EXPTIME [17]	$\mathsf{GF}^2 + TG$ – 2EXPTIME [17]+*
$\mathsf{GF} + \overrightarrow{TG}$ – 2EXPTIME [5,17]	$\mathsf{GF}^2 + \overrightarrow{TG}$ – EXPSPACE [9]

2. J. Balcázar, J. Diaz, J. Gabarró. *Structural Complexity II.* Springer 1990.
3. A. Chandra, D. Kozen, L. Stockmeyer. Alternation. *Journal of the ACM*, 28 (1), pages 114-133, 1981
4. H. Ganzinger, C. Meyer, M. Veanes. The Two-Variable Guarded Fragment with Transitive Relations. *Proc. of 14-th IEEE Symp. on Logic in Computer Science (LICS)*, pages 24-34, 1999.
5. E. Grädel. On the Restraining Power of Guards. *Journal of Symbolic Logic* 64:1719-1742, 1999.
6. E. Grädel, M. Otto, E. Rosen. Undecidability Results on Two-Variable First-Order Logic. *Bulletin of Symbolic Logic*, 3(1), pages 53-69, 1997.
7. E. Grädel, P. Kolaitis, M. Vardi. On the Decision Problem for Two-Variable First Order Logic. *Bulletin of Symbolic Logic*, 3(1), pages 53-96, 1997.
8. E. Grädel, I. Walukiewicz. Guarded Fixpoint Logic. *Proc. of 14-th IEEE Symp. on Logic in Computer Science (LICS)*, pages 45-54, 1999.
9. E. Kieroński. EXPSPACE-Complete Variant of Guarded Fragment with Transitivity. *Proc. of 19-th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 608-619, 2002.
10. D. Kozen. Results on the Propositional μ-Calculus. *Theoretical Computer Science* 27, pages 333-354, 1983.
11. O. Kupferman, A. Pnueli. Once and For All. *Proc. of 10-th IEEE Symp. on Logic in Computer Science (LICS)*, pages 25-35, 1995.
12. H. R. Lewis. Complexity Results for Classes of Quantificational Formulas. *Journal of Computer and System Sciences*, 21, pages 317-353, 1980.
13. G. De Giacomo, F. Masacci. Tableaux and Algorithms for Propositional Dynamic Logic with Converse. *Proc. of 13-th Int. Conf. on Automated Deduction (CADE)*, pages 613-627, 1996.
14. M. Mortimer. On Languages with Two Variables. *Zeitschr. f. Logik und Grundlagen d. Math.*, 21, pages 135-140, 1975.
15. V. R. Pratt. Models of Program Logics. *Proc. of 20-th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 115-122, 1979.
16. M. O. Rabin. Decidability of Second-Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141, pages 1-35, 1969.

17. W. Szwast, L. Tendera. On the Decision Problem for the Guarded Fragment with Transitivity. *Proc. of 16-th IEEE Symp. on Logic in Computer Science (LICS)*, pages 147-156, 2001.
18. M. Y. Vardi. Reasoning about The Past with Two-Way Automata. *Proc. of 25-th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 628-641, 1998.

A Game Semantics of Linearly Used Continuations

James Laird

COGS, University of Sussex, UK
jiml@cogs.susx.ac.uk

Abstract. We present an analysis of the "linearly used continuation-passing interpretation" of functional languages, based on game semantics. This consists of a category of games with a coherence condition on moves — yielding a fully abstract model of an affine type-theory — and a *syntax-independent* and *full* embedding of a category of HO-style "well-bracketed" games into it. We show that this embedding corresponds precisely to linear CPS interpretation in its action on a games model of the call-by-value (untyped) λ-calculus, yielding a proof of *full abstraction* for the associated translation.

1 Introduction

Continuation-passing-style (CPS) interpretation is widely used for reasoning about typed and untyped functional languages. However, a limitation of the standard CPS interpretation is its failure to capture the constraints on control flow which typically exist in such languages. This is because continuations become first-class objects in the *target* language which may be duplicated and discarded like any other arguments, although this corresponds to behaviour in the *source* language only if the latter also treats continuations as first-class objects (using a construct such as **call/cc**). If not, the target language contains "junk" contexts which can break equivalences which hold in the source language. One solution to this problem is a finer-grained analysis of CPS using *linear types* to control the duplication of continuations. This paper is a semantic investigation of such a "linear CPS interpretation"[1] with the object of establishing its completeness as a basis for reasoning about program equivalence.

Berdine et. al. [6] have given linear CPS translations of the call-by-value λ-calculus — and other control features such as exceptions, jumps and coroutines — into a linear λ-calculus. Syntactic mehods have been used to show that these translations are complete (in the sense that the target language does not contain any "junk" at translated types) [13, 7, 9] but these rely either on the restriction

[1] Notwithstanding the fact that, as emphasized in [6], "it is continuation transformers rather than the continuations which are linear" it seems reasonable to refer to a *linear CPS interpretation*, meaning a continuation-passing-interpretation based on linear types.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 313–327, 2003.

to a simply typed source language, or on a heavy restriction of the types in the target language. We shall achieve a more general result by semantic means.

Game semantics has been used to give models of both purely functional languages [4, 10, 16, 3] and non-functional features, including continuations [11], which are free of "junk" and hence *fully abstract*. Another useful feature of games is that intensional phenomena such as control flow and linear use of resources can be represented concretely in terms of behavioural constraints.

1.1 Contribution

The primary aim of this paper is to give a (game) semantics of linear CPS translation. By this we mean the following.

Definition 1. *Let $\mathcal{L}_1, \mathcal{L}_2$ be programming languages, and $\overline{(\_)} : \mathcal{L}_1 \to \mathcal{L}_2$ a translation between them. A semantic interpretation of $\overline{(\_)}$ consists of models $\mathcal{M}_1$ (of $\mathcal{L}_1$) and $\mathcal{M}_2$ (of $\mathcal{L}_2$), and a map $\phi : \mathcal{M}_1 \to \mathcal{M}_2$ such that for all terms M of $\mathcal{L}_1$, $\phi(\llbracket M \rrbracket_{\mathcal{M}_1}) = \llbracket \overline{M} \rrbracket_{\mathcal{M}_2}$.*

However, these formal requirements fail to capture the semantic character of a satisfactory interpretation; given a model $\mathcal{M}_2$ of $\mathcal{L}_2$, we can use a translation to determine its own interpretation by defining $\llbracket M \rrbracket_{\mathcal{M}_1} = \llbracket \overline{M} \rrbracket_{\mathcal{M}_2}$, so that $\phi : \mathcal{M}_1 \to \mathcal{M}_2$ is just an inclusion. By contrast, what we seek is a semantic interpretation in which both the models, and the mapping between them, are defined independently of the syntax.

We shall define a category of "coherence games", $\mathcal{GC}$, in which we give a fully abstract model of a target language for linear CPS translation, λ_{Aff} (a recursively typed, dual affine/non-linear λ-calculus similar to those used in [13, 6]). We shall then give a semantic interpretation of the linear CPS translation as an embedding into $\mathcal{GC}$ of a standard HO-style category of *well-bracketed* games, $\mathcal{WB}$. Specifically, we shall show that if $\overline{(\_)}$ is the linear CPS translation of λ_v (the untyped call-by-value λ-calculus) into λ_{Aff}, and $\llbracket \_ \rrbracket_{\mathcal{WB}}$ is a standard semantics of λ_v in $\mathcal{WB}$ (described in [16]) obtained by solving the domain equation $D = D \Rightarrow D_\perp$ then the following square commutes:

$$\begin{array}{ccc} \lambda_v & \xrightarrow{\llbracket \_ \rrbracket_{\mathcal{WB}}} & \mathcal{WB} \\ \Big\downarrow{\scriptstyle \overline{(\_)}} & & \Big\downarrow{\scriptstyle \phi} \\ \lambda_{\mathrm{Aff}} & \xrightarrow{\llbracket \_ \rrbracket_{\mathcal{GC}}} & \mathcal{GC} \end{array}$$

We will prove completeness of the linear CPS interpretation from a semantic perspective by showing that ϕ is *full*. Moreover, fullness will be used to prove a syntactic completeness result — *full abstraction* — for the translation of λ_v into λ_{Aff}. This can be seen as a version of the "no-junk" condition studied in [7] — we show that the translation introduces no *observable* (i.e. finite) junk.

Definition 2. *A translation* $\overline{(\_)} : \mathcal{L}_1 \to \mathcal{L}_2$ *is equationally* fully abstract *if for all terms* M_1, M_2 *of* $\mathcal{L}_1$*,* M_1 *is observationally equivalent to* M_2 *if and only if* $\overline{M_1}$ *is observationally equivalent to* $\overline{M_2}$*.*

In terms of game semantics, our interpretation can be seen as an analysis of the *bracketing condition* [4, 10]. This is known to correspond to *local control flow* in games. More precisely, relaxing it leads to models of functional languages with non-local control operators such as **call/cc** [11] which are equivalent to models constructed indirectly by CPS interpretation [12]. A connection between bracketing and linearity is evident in one direction; in a well-bracketed sequence, every question has at most one answer. More surprising is a result proved here — that under appropriate conditions, an innocent strategy which answers each question at most once must be well-bracketed.

2 An Affine Target Language for CPS

Despite being chosen independently, the linear λ-caluli used as target languages for the linear CPS translations in [13] and [6] are very similar; both are presented using dual contexts in a similar style to Barber's DILL [5]. We shall study an *affine* version as this is simpler to model and retains the key property that the affine CPS translation of λ_v is fully abstract.

In the terminology of [6], we shall restrict our attention to λ_{Aff} with only *pointed* types (this means, in essence, without sums, including atomic datatypes). However it is straightforward to extend our semantics of λ_{Aff} to include sum types, using the $\mathsf{Fam}(\mathcal{C})$ construction [3]. So types are generated from type-variables together with a single ground type R — the "answer type" of the CPS translation — by the connectives $\Rightarrow, \multimap, \&$ (intuitionistic implication, linear implication, and linear additive product) and a least fixedpoint operator.

$$T ::= X \mid \mathrm{R} \mid T \multimap T \mid T \& T \mid T \Rightarrow T \mid \mu X.T$$

Using $\Rightarrow$ instead of ! to introduce non-linear behaviour results in a much less complex calculus — terms represent derivations in the negative fragment intuitionistic natural deduction (with no "commuting conversions"), but with a finer-grained type-system.

Terms-in-context of λ_{Aff} have the form $\Gamma; \Sigma \vdash M : T$ — i.e. there are two 'zones' (multisets of typed variables) to the left of the turnstyle of which the first is intuitionistic, and the second is affine. The term-language itself is just the λ-calculus with pairing. Unlike [6], a single, standard notation for λ-abstraction and application is used for the introduction and elmination of both $\Rightarrow$ and $\multimap$, so the type which can be assigned to a term-in-context is not unique. As in [6] we take the "equality approach" [1] to recursive types — i.e. we let type-equality be the least congruence on types such that $\mu X.T = T[\mu X.T/X]$ and extend the typing rules in Table 1 with:

$$\frac{\Gamma; \Delta \vdash M : S}{\Gamma; \Delta \vdash M : T} S = T$$

$$\frac{}{\Gamma,x{:}T;\Delta\vdash x{:}T} \qquad \frac{}{\Gamma;\Delta,x{:}T\vdash x{:}T}$$

$$\frac{\Gamma;\Delta,x{:}S\vdash M{:}T}{\Gamma;\Delta\vdash\lambda x.M{:}S\multimap T} \qquad \frac{\Gamma;\Delta\vdash N{:}S \quad \Gamma;\Delta'\vdash M{:}S\multimap T}{\Gamma;\Delta,\Delta'\vdash M\ N{:}T}$$

$$\frac{\Gamma,x{:}S;\Delta\vdash M{:}T}{\Gamma;\Delta\vdash\lambda x.M{:}S\Rightarrow T} \qquad \frac{\Gamma;\_\vdash N{:}S \quad \Gamma;\Delta\vdash M{:}S\Rightarrow T}{\Gamma;\Delta\vdash M\ N{:}T}$$

$$\frac{\Gamma;\Delta\vdash M{:}S \quad \Gamma;\Delta\vdash N{:}T}{\Gamma;\Delta\vdash\langle M,N\rangle{:}S\&T} \qquad \frac{\Gamma;\Delta\vdash M{:}T_1\&T_2}{\Gamma;\Delta\vdash\pi_i(M){:}T_i}$$

Table 1. Typing judgements for λ_{Aff}

Definition 3. *The equational theory of λ_{Aff}, $=_{\beta\eta\pi}$ is generated by the rules:*

$$(\beta)\quad (\lambda x.M)\ N =_{\beta\eta\pi} M[N/x]$$
$$(\eta)\quad \lambda x.(M\ x) =_{\beta\eta\pi} M,\ x\notin FV(t)$$
$$(\pi)\quad \pi_i(\langle M_1,M_2\rangle) =_{\beta\eta\pi} M_i,\ i=1,2$$
$$(\pi_\eta)\quad \langle\pi_1(M),\pi_2(M)\rangle =_{\beta\eta\pi} M$$

We shall not give an explicit operational semantics of λ_{Aff}, but a notion of convergence based on the existence of a head-normal form

Definition 4. *The head-normal forms of λ_{Aff} are given by the grammar:*
$H ::= B \mid \lambda x.H \mid \langle H,H\rangle$*, where:*
$B ::= x \mid B\ M \mid \pi_i(B)$.
For a closed term M, we shall write $M\Downarrow$ if there exists H such that $M =_{\beta\eta\pi} H$.

There is no canonical notion of observational equivalence for λ_{Aff}, because there is no canonical notion of observation. Basically, we have a choice between observing convergence to head-normal form at either linear or non-linear function type, and these are *not* equivalent in general. In the context of the linear CPS translation, it is the former which is most suitable, since it corresponds to the observations available in the source language. (Allowing observations at non-linear function types causes failure of full abstraction for the translation.)

Definition 5. *Let $\simeq_L$ be contextual equivalence of λ_{Aff} terms defined with respect to observations at the linear type* $\mathrm{R}\multimap\mathrm{R}$ *— i.e. if $M,N:T$, then $M\simeq_L N$ if for all closing contexts $C[\_]:\mathrm{R}\multimap R$, $C[M]\Downarrow$ if and only if $C[N]\Downarrow$.*

The source language for our example of linear CPS translation will be the untyped call-by-value λ-calculus, λ_v, which has terms given by the grammar:
$M ::= x \mid \lambda x.M \mid M\ N$,
and an operational semantics given by:

$$\frac{}{\lambda x.P\Downarrow\lambda x.P} \qquad \frac{M\Downarrow\lambda x.P \quad N\Downarrow\lambda y.Q \quad P[\lambda y.Q/x]\Downarrow V}{M\ N\Downarrow V}$$

A standard CPS translation of λ_v can be given an affine typing in λ_{Aff} [6].

Definition 6. *Let* $D = \mu X.(X \Rightarrow (X \Rightarrow \mathrm{R}) \multimap \mathrm{R})$. *A term* M *of* λ_v *with free variables* $x_1, \ldots x_n$ *is interpreted as a* λ_{Aff} *term* $x_1 : D, \ldots, x_n : D; \_ \vdash \overline{M} : (D \Rightarrow \mathrm{R}) \multimap \mathrm{R}$ *defined as follows:*

- $\overline{x} = \lambda k.k\, x$,
- $\overline{\lambda x.M} = \lambda k.k\, (\lambda x.\overline{M})$,
- $\overline{M\, N} = \lambda k.\overline{M}\, (\lambda m.\overline{N}\, (\lambda n.(m\, n)\, k))$.

3 Game Semantics

We shall now present a category of "Hyland-Ong-style" games, containing a semantics of λ_{Aff}, which is universal for finite types. It is based on the addition of a notion of *coherence* to HO arenas in the form of a symmetric relation on moves, and a corresponding refinement of the notion of legal sequence. A second significant feature is that we drop the visibility condition from plays; this actually cuts down the space of innocent and coherent strategies, allowing our universality result to be proved.[2]

First, we shall briefly describe standard notions of *underlying* arenas and innocent strategies which can form a basis for both coherence games (by adding a coherence relation) or the standard "well-bracketed" games (by adding a question/answer labelling to moves).

Definition 7. *An arena* A *is a pair* $\langle M_A, \vdash_A \rangle$ *where* $\vdash_A \subseteq (M_A \cup \{*\}) \times M_A$ *(the* enabling *relation) allows a unique* polarity *to be* inferred *for each move, according to the following rules:*
m is an O-move if it is initial (i.e. $* \vdash a$*), or is enabled by some P-move.*
m is a Player (P)-move if it is enabled by some O-move.

A *justified sequence* over an arena A is a sequence of elements of M_A in which each non-initial move a comes with a pointer to a preceding, enabling move $j_s(a)$. A legal sequence on A is a justified sequence which is *alternating* — Player moves follow Opponent moves and vice-versa. The product and function-space constructions on arenas are standard [10, 16].

Definition 8. *For arenas* A_1, A_2*, define:*

- $A_1 \times A_2 = \langle M_{A_1} + M_{A_2}, \{\langle\langle m, i\rangle, \langle n, i\rangle\rangle \mid m \vdash_{A_i} n\} \cup \{\langle *, \langle m, i\rangle\rangle \mid * \vdash_{A_i} m\}\rangle$,
- $A_1 \to A_2 = \langle M_{A_1} + M_{A_2}, \{\langle\langle m, i\rangle, \langle n, i\rangle\rangle \mid m \vdash_{A_i} n\} \cup \{\langle\langle m, 2\rangle, \langle n, 1\rangle\rangle \mid * \vdash_{A_2} m \wedge * \vdash_{A_1} n\} \cup \{\langle *, \langle m, 2\rangle\rangle \mid * \vdash_{A_2} m\}\rangle$.

The empty arena, with no moves, will be written $\mathbf{1}$.

An innocent strategy will be represented as a set of *Player views* [16].

[2] This departure from existing games models of linear logic is necessary — in particular, although the AJM game semantics of PCF [4] is based on a model of intuitionistic affine type theory which can be used to model λ_{Aff}, this semantics is not fully complete.

Definition 9. *The* Player view *of a non-empty justified sequence s is a sequence with justification pointers, $\ulcorner s \urcorner$, defined by induction as follows:*
$\ulcorner sa \urcorner = \ulcorner s \urcorner a$, *if a is a Player move,*
$\ulcorner sa \urcorner = a$, *if a is an initial Opponent move,*
$\ulcorner satb \urcorner = \ulcorner s \urcorner ab$, *if b is an Opponent move justified by a.*
A legal P-view is a legal sequence s such that $\ulcorner s \urcorner = s$.

Definition 10. *An innocent strategy is a set of even-length legal P-views which is non-empty, even-prefix-closed and even-branching — i.e. $sab, sac \in \sigma$ implies $b = c$.*

From an innocent strategy we can obtain a strategy in the standard form of an even-prefix-closed set of even-length legal sequences by taking its *view-closure*.

Definition 11. *Given a strategy $\sigma : A$, the "view closure" $VC(\sigma)$ is defined to be the least set of legal sequences on A such that $\varepsilon \in VC(\sigma)$, and if $s \in VC(\sigma)$ and $\ulcorner sab \urcorner \in \sigma$, then $sab \in VC(\sigma)$.*

Composition of strategies is by taking the views of the "parallel composition plus hiding" of ther view-closures.

Definition 12. *For $\sigma : A_1 \to A_2, \tau : A_2 \to A_3$: $\sigma; \tau = \{\ulcorner t \upharpoonright A_1, A_3 \urcorner \mid t \upharpoonright A_1, A_2 \in VC(\sigma) \wedge t \upharpoonright A_2, A_3 \in VC(\tau)\}$.*

3.1 Coherence Arenas

Given an arena A, we say that moves $m, n \in M_A$ are *co-enabled* if $\exists l \in (M_A)_*.(l \vdash_A m) \wedge (l \vdash_A n)$.

Definition 13. *A coherence arena (or C-arena) A is a pair $\langle |A|, \sim_A \rangle$ consisting of an underlying arena $|A|$ together with a symmetric relation $\sim_A$ between co-enabled moves of $|A|$.*
A coherent sequence of A is a legal sequence s of $|A|$ which satisfies the following conditions:

- *All initial moves in s are coherent: if $ta, t'a' \sqsubseteq s$ and $* \vdash a, b$ then $a \sim_A b$.*
- *Any two non-initial moves in s with the same justifier are coherent: if $ta, t'a' \sqsubseteq s$ and $j_s(a) = j_s(b)$ then $a \sim_A b$.*

We can now define notions of additive and multiplicative product; both are based on the product of the underlying arenas, but they are differentiated by varying the coherence relations on initial moves. This distinction can be summarised: in $A \otimes B$ every initial move of A is coherent with every initial move of B, whereas in $A\&B$ every initial move of A is incoherent with every initial move of B. Hence a coherent sequence of $A \otimes B$ consists of a pair of interleaved sequences of A and B, and a coherent sequence of $A\&B$ is a sequence wholly from A or wholly from B.

Definition 14. *Given C-Arenas A_1, A_2, define the following C-arenas:*

Tensor Product $A_1 \otimes A_2 = \langle |A_1| \times |A_2|, \sim_{A_1 \otimes A_2} \rangle$ *where* $\langle m, i \rangle \sim_{A_1 \otimes A_2} \langle n, j \rangle$ *iff* $i = j$ *implies* $m \sim_{A_i} n$.
Additive Product $A_1 \& A_2 = \langle |A_1| \times |A_2|, \sim_{A_1 \& A_2} \rangle$ *where* $\langle m, i \rangle \sim_{A_1 \& A_2} \langle n, j \rangle$ *iff* $i = j$ *and* $m \sim_{A_i} n$.
Linear Function Space $A_1 \multimap A_2 = \langle |A_1| \to |A_2|, \sim_{A_1 \multimap A_2} \rangle$, *where* $\langle m, i \rangle \sim_{A_1 \multimap A_2} \langle n, j \rangle$ *iff* $i = j$ *implies* $m \sim_{A_i} n$.

We shall say that a strategy is *coherent* if it is never the first participant in a dialogue to violate the coherence condition.

Definition 15. *For any C-arena A, an innocent strategy on A is coherent if whenever $sa \in VC(\sigma)$ and s is coherent then sa is coherent.*

However, we cannot define a category of C-arenas in the standard fashion by taking morphisms from A to B to be coherent strategies on $A \multimap B$ fails; the composition of coherent strategies is not coherent (see [12]). To solve this problem, we place a further restriction on the coherent strategies which are permitted as morphisms.

Definition 16. *A C-strategy from A to B is a coherent strategy on $A \multimap B$ such that if $s \in VC(\sigma)$ is coherent, then every two moves in s which are initial in A are coherent — i.e. if $ta, t'a' \sqsubseteq s$ and $* \vdash_A a, b$ then $a \sim_A b$.*

It is now straightforward to show that the composition of innocent C-strategies is an innocent C-strategy and so we can define category $\mathcal{GC}$ with C-arenas as objects and C-strategies from A to B as morphisms from A to B.

Proposition 1. $(\mathcal{GC}, \mathbf{1}, \otimes)$ *is a symmetric monoidal category with a cartesian product, &.*

The price we have paid is the loss of the symmetric monoidal *closed* structure — $\mathcal{GC}$ does not have all exponentials in the following sense.

Definition 17. *Let A, B be objects in a symmetric monoidal category $(\mathcal{C}, I, \otimes)$. An exponential of B by A is a an object B^A such that for all C in $\mathcal{C}$, there is an isomorphism:* $\mathrm{ev}_{A,B} : \mathcal{C}(C \otimes A, B) \cong \mathcal{C}(C, B^A)$ *which is natural in C.*

We have an isomorphism of arenas — $(A \otimes B) \multimap C \cong A \multimap (B \multimap C)$ — but not, in general $\mathcal{GC}(A \otimes B, C) \cong \mathcal{GC}(A, B \multimap C)$. However, to model λ_{Aff}, we do not require that B^A exists for any arena B, but only for B within a specified collection of *well-opened* arenas, for which the notions of coherent strategy and C-strategy coincide.

Definition 18. *Say that a C-arena A is* well-opened *if for any pair of initial moves $m, n \in M_A$ (not necessarily distinct), $m \not\sim_A n$. We shall write $\mathcal{WO}$ for the full subcategory of $\mathcal{GC}$ consisting of well-opened C-arenas.*

Lemma 1. *If B is well-opened, then for any C-arena A, $A \multimap B$ is an exponential of B by A.*

Proof. If D is any C-arena, then *every* coherent strategy σ on $D \multimap B$ is a C-strategy from D to B, since by well-openedness of B, every coherent sequence on $D \multimap B$ contains at most one initial move, and so any two initial moves of D in s must have the same justifier, and hence be coherent. So for any arena C, $\mathcal{GC}(C \otimes A, B) \cong \mathcal{GC}(\mathbf{1}, (C \otimes A) \multimap B) \cong \mathcal{GC}(\mathbf{1}, C \multimap (A \multimap B)) \cong \mathcal{GC}(C, A \multimap B)$ as required.

Lemma 2. *If B is well-opened, then for any A, $A \multimap B$ is well-opened, and if A and B are well opened, then $A\&B$ is well-opened.*

We can now use coherence to define a monoidal co-monad $! : \mathcal{GC} \to \mathcal{GC}$. The C-arena $!A$ has the same underlying arena as A, but all of the "incoherences" between the initial moves of A are removed.

Definition 19. *For any C-arena A, define $!A = \langle |A|, \sim_{!A} \rangle$, where $m \sim_{!A} n$, if $m \sim_A n$ or $* \vdash m, n$.*

Thus for a well-opened arena A, the coherent sequences of $!A$ consist of multiple interleaved sequences ("threads") of A. The action of ! on morphisms is simple: if $\sigma : !A \to B$ is a coherent strategy, then $\sigma^\dagger : !A \to !B$ has the same underlying strategy. For each C-arena A we have $\mathtt{der}_A : !A \to A$ which has the same underlying strategy as the identity. We also have equalities of arenas: $!(A\&B) = !A \otimes !B = !(A \otimes B)$ yielding the monoidal natural transformations and contraction maps $\mathtt{con}_A : !A \to !A \otimes !A$ for each A. Thus we can define $A \Rightarrow B = !A \multimap B$.

3.2 Interpretation of $\boldsymbol{\lambda_{\mathbf{Aff}}}$ in $\mathcal{GC}$

We have defined functors $\_\&\_ : \mathcal{WO} \times \mathcal{WO} \to \mathcal{WO}$, $\_ \multimap \_ : \mathcal{WO}^{OP} \times \mathcal{WO} \to \mathcal{WO}$ and $\_ \Rightarrow \_ : \mathcal{WO}^{OP} \times \mathcal{WO} \to \mathcal{WO}$ which we shall use to interpret the corresponding type-constructors in λ_{Aff}. To interpret R, we must identify a well-opened answer-object, and to eliminate "junk" we choose the smallest such non-terminal arena.

Definition 20. *Let o be the arena with one move which is not coherent with itself, i.e. $M_o = \{o\}$, $\vdash_o = \langle *, o \rangle$ and $o \not\sim_o o$.*

To interpret recursive types in $\mathcal{GC}$, we use the "information-system"-like approach studied in depth in [16], which allows domain equations to be solved up to equality, as required by λ_{Aff}. First, we define an inclusion order on C-arenas.

Definition 21. *$A_1 \trianglelefteq A_2$ if:*

- $M_{A_1} \subseteq M_{A_2}$,
- $\vdash_{A_1} = \vdash_{A_2} \cap ((M_{A_1})_* \times M_{A_1})$
- $\sim_{A_1} = \sim_{A_2} \cap (M_{A_1} \times M_{A_1})$.

If $A \trianglelefteq B$ then there are obvious inclusion and projection maps $A \to^{\mathtt{in}} B \to^{\mathtt{out}} A$ such that $\mathtt{in};\mathtt{out} = \mathsf{id}_{\mathtt{A}}$ and $\mathtt{out};\mathtt{in} \subseteq \mathsf{id}_{\mathtt{B}}$.

Proposition 2. *C-Arenas form a large cpo, ordered by* $\trianglelefteq$.

Each of the functors $\&, \multimap$ and $!$ are continuous with respect to $\trianglelefteq$. Thus we can define a least fixed point operator by iteration.

Definition 22. *Given a continuous functor* $F : \mathcal{WO}^{OP} \times \mathcal{WO} \to \mathcal{WO}$, *let* $\Delta(F) \in \mathcal{WO} = \bigsqcup_{i \in \omega} F^i(\mathbf{1})$.

Then $F(\Delta(F), \Delta(F)) = \Delta(F)$ and so we have a sound model of λ_{Aff}, based on the following interpretations of types with n free variables as mixed-variance functors from $(\mathcal{WO}^{OP} \times \mathcal{WO})^n$ to $\mathcal{WO}$:

$$\begin{array}{cc} [\![\mathrm{R}]\!] = o & [\![X]\!] = \pi_r \\ [\![S \multimap T]\!] = [\![S]\!] \multimap [\![T]\!] & [\![S \Rightarrow T]\!] = ![\![S]\!] \multimap [\![T]\!] \\ [\![S \& T]\!] = [\![S]\!] \& [\![T]\!] & [\![\mu X.T(X, Y_1, \ldots Y_n)]\!] = \Delta([\![T(\cdot, Y_1, \ldots, Y_n)]\!]) \end{array}$$

Each term-in-context $x_1 : S_1, \ldots, x_m : S_m; y_1 : T_1, \ldots y_n : T_n \vdash M : U$ is thus interpreted as a natural transformation from $![\![S_1]\!] \otimes \ldots \otimes ![\![S_k]\!] \otimes [\![T_1]\!] \otimes \ldots \otimes [\![T_m]\!]$ to $[\![U]\!]$ following e.g. [5]. (We shall write $[\![S_1, \ldots, S_n; T_1, \ldots T_m \vdash U]\!]$ for the set of such natural transformations.)

Using approximation relations as in [17, 16], we prove the following computational adequacy result.

Proposition 3. *For any program* M *of* λ_{Aff}, $M \Downarrow$ *if and only if* $[\![M]\!] \neq \bot$.

We also have a universality result for *finite types*, and hence a full abstraction result with respect to an intrinsic preorder.

Definition 23. *The (closed) finite types of* λ_{Aff} *are generated by the grammar:* $F ::= \mathrm{R} \mid F \& F \mid F \multimap F \mid F \Rightarrow F$

For an innocent strategy σ, let $\#(\sigma)$ be the cardinality of σ as a set of views, so we may say that σ is finite if $\#(\sigma)$ is finite.

Proposition 4. *Let* F *be a finite type, and* Γ, Δ *contexts of finite types. Then every finite, innocent C-strategy* $\sigma \in [\![\Gamma; \Delta \vdash F]\!]$ *is definable — i.e. there exists a* λ_{Aff}*-term-in-context* $\Gamma; \Delta \vdash M_\sigma : F$ *such that* $\sigma = [\![M_\sigma]\!]$.

Proof. (Sketch) We apply an inductive decomposition similar to the decomposition theorem for the simply-typed λ-calculus, described axiomatically in [2]. Formally, proof is by induction on $\#(\sigma)$. If this is zero then σ is the empty strategy, which is definable as a divergent program. We prove the inductive case by a series of lemmas, all of which are implicitly dependent on the inductive hypothesis. The first is a subcase for *strict* strategies[3] and π*-atomic* types, which are given by the following grammar:
$P ::= \mathrm{R} \mid F \multimap P \mid F \Rightarrow P$.
We shall write π-atomic types in the form $S_1 \Rightarrow \ldots \Rightarrow S_m \Rightarrow T_1 \multimap \ldots \multimap (T_n \multimap \mathrm{R})$, or $\boldsymbol{S} \Rightarrow (\boldsymbol{T} \multimap \mathrm{R})$.

[3] A strategy in $\mathcal{GC}(A, B)$ is *strict* if the first P-move in σ (if any) is in A.

Lemma 3. *If P_1, P_2 are π-atomic types and $\sigma \in [\![\_; P_1 \vdash P_2]\!]$ is strict then σ is definable.*

Proof. By induction on the size of P_2. The base case is $P_2 = \mathrm{R}$ and $P_1 = \boldsymbol{S} \Rightarrow (\boldsymbol{T} \multimap \mathrm{R})$ — using the induction hypothesis on $\#(\sigma)$ we can find $\boldsymbol{M} : \boldsymbol{S}$ and $\boldsymbol{N} : \boldsymbol{T}$ such that $\sigma = [\![x : P_1 \vdash ((x\ \boldsymbol{M})\ \boldsymbol{N}) : \mathrm{R}]\!]$. The induction cases are:

- If $P_2 = C \Rightarrow P$, then by induction (on P_2) we can find a term $\_; x : (\boldsymbol{C \Rightarrow S}) \Rightarrow ((\boldsymbol{C \Rightarrow T}) \multimap \mathrm{R}) \vdash M : P$ such that $\sigma = [\![y : \boldsymbol{S} \Rightarrow (\boldsymbol{T} \multimap \mathrm{R}) \vdash \lambda z.M[(\lambda \boldsymbol{uv}.(y\ (\boldsymbol{u}\ \boldsymbol{z})\ (\boldsymbol{v}\ \boldsymbol{z})))/x] : C \Rightarrow P]\!]$
- If $P_2 = C \multimap P$ then there exists $i \leq m$ and $x : \boldsymbol{S} \Rightarrow (\boldsymbol{T'} \multimap \mathrm{R}) \vdash M : P$ (where $T'_i = C \multimap T_i$ and $T'_j = T_j$ for $j \neq i$) such that $\sigma = [\![\_; y : \boldsymbol{S} \Rightarrow (\boldsymbol{T} \multimap \mathrm{R}) \vdash \lambda z.M[(\lambda \boldsymbol{u}.\lambda \boldsymbol{v}.y\ \boldsymbol{u} v_1 \dots v_{i-1}(v_i\ z) v_{i+1} \dots v_m)/x] : C \multimap P]\!]$

Lemma 4. *If F is any finite type, P is a π-atomic type, and $\sigma \in [\![\_; F \vdash P]\!]$ is strict then σ is definable.*

Proof. is by induction on the size of F. If this is an atomic formula then Lemma 3 applies. Otherwise $F = \boldsymbol{S} \Rightarrow (\boldsymbol{T} \multimap U_1 \& U_2)$ and we can find $i \in \{1, 2\}$ and a term-in-context $x : \boldsymbol{S} \Rightarrow (\boldsymbol{T} \multimap U_i) \vdash M : P]\!]$ such that $\sigma = [\![y : \boldsymbol{S} \Rightarrow (\boldsymbol{T} \multimap U_1 \& U_2) \vdash M[\lambda \boldsymbol{u}.\lambda \boldsymbol{v}.\pi_i(y\ \boldsymbol{uv})/x] : P]\!]$.

Lemma 5. *If $\sigma \in [\![S_1, \dots, S_m; T_1, \dots, T_n \vdash \mathrm{R}]\!]$ is strict then σ is definable.*

Proof. If the first P-move in σ is in some $[\![S_i]\!]$, then using Lemma 4 we can find a term $\_; x_i : S_i \vdash M : \boldsymbol{S} \Rightarrow \boldsymbol{T} \multimap \mathrm{R}$ such that $\sigma = [\![x_1 : S_1, \dots, x_m : S_m; y_1 : B_1, \dots, y_n : B_n \vdash (M\ \boldsymbol{x})\ \boldsymbol{y} : \mathrm{R}]\!]$.
If the first P-move in σ is in some $[\![T_j]\!]$, then we can find a term $\_; y : T_j \vdash M : \boldsymbol{S} \Rightarrow T_1 \multimap \dots \multimap T_{j-1} \multimap T_{j+1} \multimap \dots \multimap T_k \multimap \mathrm{R}$ such that $\sigma = [\![x_1 : S_1, \dots, x_n : S_m; y_1 : T_1, \dots, y_n : T_n \vdash (M\ \boldsymbol{x})\ y_1 \dots y_{j-1} y_{j+1} \dots y_k : \mathrm{R}]\!]$.

Finally, we can complete the induction to prove Proposition 4 by showing that if $\sigma \in [\![\Gamma; \Delta \vdash F]\!]$ then σ is definable, by induction on the size of F. If $F = \mathrm{R}$, then σ must be strict, and Lemma 5 applies. Otherwise $F = F_1 \& F_2$, or $F = F_1 \multimap F_2$ or $F = F_1 \Rightarrow F_2$ and can be decomposed and reconstructed by projection/pairing or uncurrying/currying.

Definition 24. *Define the* intrinsic equivalence $\approx$ *on strategies $\sigma, \tau : A$:*
$\sigma \approx \tau$ if for all C-strategies $\rho : A \to (o \multimap o)$, $\sigma; \rho = \bot$ if and only if $\tau; \rho = \bot$.

The proof of the full abstraction follows a standard pattern [4, 10, 16], based on Proposition 4 and the fact that for any closed λ_{Aff} type, T there is a chain of finite types $F_1, F_2, \dots$ such that $[\![T]\!] = \bigsqcup_{i \in \omega} [\![F_i]\!]$.

Theorem 1 (Full Abstraction). *For all λ_{Aff}-terms $M, N : T$, $M \simeq_L N$ if and only if $[\![M]\!] \approx [\![N]\!]$.*

4 Well-Bracketing and Linear CPS

We can now show that a variant of the original category of HO games [10] (and its extension by McCusker with lifted sums [16]) can be fully embedded in the category of coherence games, and that this embedding preserves denotations with respect to linear CPS translation.

Definition 25. *A bracketed arena (or B-arena) is a pair* $\langle |A|, \lambda_A \rangle$ *consisting of an underlying arena* $|A|$ *(as per Definition 7), with a labelling function* $\lambda_A : M_A \to \{Q, A\}$*, which partitions the set of moves into* questions *and* answers*. Answers must be enabled (and must only be enabled) by questions.*

The product and function-space of bracketed arenas are based on the corresponding constructions on the underlying arenas — i.e. $A \times B = \langle |A| \times |B|, [\lambda_A, \lambda_B] \rangle$ and $A \Rightarrow B = \langle |A| \to |B|, [\lambda_A, \lambda_B] \rangle$. The rôle of answer labelling is to allow the definition of the *bracketing condition.*

Definition 26. *For each justified sequence,* s*, we define a prefix* $\mathsf{pending}(s) \sqsubseteq s$ *as follows:*
$\mathsf{pending}(\varepsilon) = \varepsilon$*,*
$\mathsf{pending}(sq) = sq$ *if* q *is a question,*
$\mathsf{pending}(sqta) = \mathsf{pending}(s)$*, if* a *is an answer justified by* q*.*

The pending question of s is the final move in $\mathsf{pending}(s)$, if any.

Definition 27. *An alternating justified sequence* s *on a bracketed arena is* well-bracketed *if every answer in* s *is justified by the pending question — i.e. if* $rqta \sqsubseteq s$ *and* a *is an answer justified by* q*, then* $rq = \mathsf{pending}(rqt)$*.*
An innocent and well-bracketed *strategy on a B-arena is an innocent strategy on the underlying arena such that whenever* $sab \in VC(\sigma)$ *and* sa *is well-bracketed then* sab *is well-bracketed.*

The following lemma, which characterizes innocent and well-bracketed strategies in terms of their P-views is proved in [11, 12].

Lemma 6. *An innocent strategy* σ *is well-bracketed if and only if for all* $s \in \sigma$*,* s *is well-bracketed.*

The composition of innocent and well-bracketed strategies (as defined in Definition 12) is well-bracketed and thus we have a cartesian closed category $(\mathcal{WB}, \mathbf{1}, \times, \Rightarrow)$ with B-arenas as objects and innocent well-bracketed strategies on $A \Rightarrow B$ as morphisms from A to B [10, 12]. $\mathcal{WB}$ also has a *lifted sum* construction [16], obtained by adding an initial question and answers corresponding to each summand. To give a semantics of λ_v, we only require the unary version — lifting.

Definition 28. *For any B-arena* A*, the lifting* $A_\perp$ *is defined as follows:*

- $M_{A_\perp} = (\{q\} \cup \{a\}) + M_A$,
- $\vdash_{A_\perp} = \{\langle *, \mathtt{inl}(q) \rangle, \langle \mathtt{inl}(q), \mathtt{inl}(a) \rangle\} \cup \{\langle \mathtt{inl}(a), \mathtt{inr}(b) \rangle \mid * \vdash_A b\} \cup \{\langle \mathtt{inr}(m), \mathtt{inr}(n) \mid m \vdash_A n\}$,
- $\lambda_{A_\perp}(\mathtt{inl}(q)) = Q, \quad \lambda_{A_\perp}(\mathtt{inl}(a)) = A, \quad \lambda_{A_\perp}(\mathtt{inr}(b)) = \lambda_A(b)$.

There is an evident operation taking $\sigma : A \Rightarrow B_\perp$ to $\sigma^* : A_\perp \Rightarrow B_\perp$, and well-bracketed strategies $\eta_A : A \to A_\perp$ and $t_{A,B} : A \times B_\perp \to (A \times B)_\perp$, making lifting a strong monad on $\mathcal{WB}$. We can solve domain equations up to equality in $\mathcal{WB}$, just as in $\mathcal{GC}$ — by taking fixed points of mixed variance functors which are minimal with respect to the inclusion order — $A \trianglelefteq B$ if $|A| \trianglelefteq |B|$ and $\lambda_A = \lambda_B \restriction M_B$. Thus a semantics of the call-by-value λ-calculus can be defined in standard fashion.

Definition 29. *Let D be the least solution to $D = D \Rightarrow D_\perp$ in $\mathcal{WB}$ — i.e. $D = \bigsqcup_{i\in\omega} F^i(\mathbf{1})$, where $F(X) = X \Rightarrow X_\perp$. For each term-in-context $x_1, \ldots x_n \vdash M$ of λ_v, let $[\![M]\!]_{\mathcal{WB}} : D^n \to D_\perp$ be the innocent, well-bracketed strategy denoting M, defined as in [16].*

Let $\Sigma = \mathbf{1}_\perp$ be the game with a single question and answer, and define the *intrinsic equivalence* $\approx$ on well-bracketed strategies $\sigma, \tau : A$ as follows:
$\sigma \approx \tau$ if for all $\rho : A \to \Sigma$, $\sigma; \rho = \perp$ if and only if $\tau; \rho = \perp$.
We can now state McCusker's full abstraction result for the model of λ_v in $\mathcal{WB}$.

Proposition 5 (McCusker [16]). *For any λ_v terms M, N: $M \simeq N$ if and only if $[\![M]\!]_{\mathcal{WB}} \approx [\![N]\!]_{\mathcal{WB}}$.*

4.1 Embedding the Well-bracketed Games into $\mathcal{GC}$

From each B-arena A we can define a C-arena $\phi(A)$ with the same underlying structure, and a coherence relation in which moves are incoherent if and only if they are answers to the same question.

Definition 30. *$\phi(A) = \langle |A|, \sim_A \rangle$, where $m \not\sim_A n$ if $\lambda_A(m) = \lambda_A(n) = A$, and $m \sim_A n$ otherwise.*

ϕ acts as the identity on underlying innocent strategies. To show that this defines a functor, we need to show that any innocent and well-bracketed strategy on a B-arena is a coherent strategy on the associated C-arena.

Lemma 7. *If sqt is a coherent sequence on $\phi(A)$ and q is a question such that $\mathsf{pending}(sqt) = \mathsf{pending}(s)$, then q is answered in t.*

Proof. is by induction on the length of sqt. Given $sqtb$ where b is an O move, we have *either* that b is a question — in which case $\mathsf{pending}(sqtb) \neq \mathsf{pending}(s)$ — *or* b is an answer to a question q' in t — i.e. $t = t'q't''b$, where $\mathsf{pending}(sqt') = \mathsf{pending}(s)$ and hence q is answered in t' by induction hypothesis — or b is an answer to a question q' in s — i.e. $s = s'q't'$, and $\mathsf{pending}(s'q't') = \mathsf{pending}(sqtb) = \mathsf{pending}(s')$ and hence q' is already answered in t' which contradicts coherence of $sqtb$.

Proposition 6. *For any B-arena A, if σ is a well-bracketed innocent strategy on A then it is a coherent strategy on $\phi(A)$.*

Proof. It is sufficient to show that if $sbc \in VC(\sigma)$ and sb is coherent, then $\mathsf{pending}(sb)$ has not already been answered and hence sbc is coherent. We show this by induction on sequence length: Given an (odd-length) sequence sb, either b is a question — in which case it is the pending question, and unanswered — or b is an answer, and $sb = s'qtb$, where b answers q and so $\mathsf{pending}(sb) = \mathsf{pending}(s')$. The pending question in sb is not answered in s by inductive hypothesis, and so the only remaining possibility is that it is answered in t. In other words there is a prefix $s'qt'a \sqsubseteq s$, in which a answers $\mathsf{pending}(s) = \mathsf{pending}(s')$. But by assumption we have $\mathsf{pending}(s'qt') = \mathsf{pending}(s')$ and hence by Lemma 7 q is answered in t', which contradicts coherence of sb.

Functoriality and injectivity of ϕ are immediate, so it remains to show *fullness* of the embedding. It is clearly not the case in general that if s is a coherent sequence on $\phi(A)$ then s is well-bracketed; the coherence condition prevents repetition of answers, but not *jumping* — answering a question out of turn. However, we can show that if σ is an innocent and coherent strategy on $\phi(A)$, then σ must be well-bracketed, as otherwise Opponent can force σ to violate coherence because of innocence. A non-well-bracketed strategy on $(o \Rightarrow o) \Rightarrow \Sigma$ which translates to a non-coherent strategy on $(o \Rightarrow o) \Rightarrow (o \multimap o)$ is depicted in Figure 1.

Proposition 7. *For any C-strategy $\sigma : \phi(A) \to \phi(B)$, σ is a well-bracketed strategy on $A \Rightarrow B$.*

Proof. By Lemma 6 it suffices to show that every $s \in \sigma$ is well-bracketed. Suppose for a contradiction that we have a minimal length sequence $sqta \in \sigma$ such that q is the pending question in sqt but a is not justified by q. Then the justified sequence $sqtaqt$ in which the two q moves have the same justifier is a coherent sequence on $\phi(A \Rightarrow B)$. But $\ulcorner sqtat \urcorner = sqt$ and hence $sqtata \in VC(\sigma)$, which violates the coherence condition.

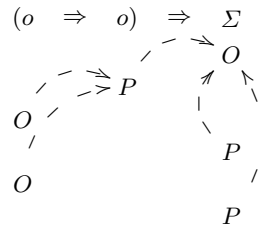


Fig. 1. A non-well-bracketed strategy forced to violate coherence

Theorem 2. *The category of bracketed arenas and innocent, well-bracketed strategies embeds fully in the category of C-arenas and C-strategies.*

It remains to observe that interpreting λ_v in $\mathcal{WB}$ followed by embedding in $\mathcal{GC}$ corresponds to interpretation via linear CPS translation. The embedding preserves the standard cartesian closed structure of $\mathcal{WB}$ — the key point is that it also respects the definitions of lifting in the following sense.

Lemma 8. *If A is a B-arena then $\phi(A_\perp) = (\phi(A) \Rightarrow o) \multimap o$.*

Hence $\phi(D) = [\![\mu X.X \Rightarrow ((X \Rightarrow \mathrm{R}) \multimap \mathrm{R}]\!]_{\mathcal{GC}}$ and it is straightforward to check that the embedding commutes with linear CPS translation as required.

Proposition 8. *For any program M of λ_v, $\phi([\![M]\!]_{\mathcal{WB}}) = [\![\overline{M}]\!]_{\mathcal{GC}}$.*

Theorem 3. *The linear CPS translation from λ_v to λ_{Aff} is fully abstract.*

Proof. Soundess is straightforward. For completeness, suppose $\overline{M} \not\simeq_L \overline{N}$. Then $[\![\overline{M}]\!]_{\mathcal{GC}} \not\simeq [\![\overline{N}]\!]_{\mathcal{GC}}$ by Theorem 1 — i.e. w.l.o.g. there exists a coherent strategy $\rho : [\![(\mu X.(X \Rightarrow X_\perp)_\perp]\!]_{\mathcal{GC}} \to \mathrm{R} \multimap \mathrm{R}$ such that $[\![\overline{M}]\!]; \rho = \perp$ and $[\![\overline{N}]\!]; \rho \neq \perp$. By Proposition 7, ρ is a well-bracketed strategy on $D_\perp \to \Sigma$ such that $[\![M]\!]_{\mathcal{WB}}; \rho = \perp$ and $[\![N]\!]_{\mathcal{WB}}; \rho \neq \perp$ so by Proposition 5 $M \not\simeq N$.

5 Conclusions and Further Directions

There are several possibilities for extending this work. On the logical side, Streicher's domain equation for the designs of Ludics [18] corresponds to a type of λ_{Aff}; the correspondence between designs and the strategies at this type seems to parallel closely the account given in [8]. There is also a close relationship between λ_{Aff}, and proofs in polarized classical linear logic with both lifting and the exponentials, and we can construct a model of the latter along the lines of [15] which is, moreover, fully complete.

As described in [6], several other control features, including exception-handling, `GOTO`-style jumps and coroutines can be given linear CPS translations in λ_{Aff}, and hence modelled in our category of games. However, apart from exception-handling (for which the linear CPS interpretation is essentially equivalent to the standard monadic interpretation using a lifted sum type) these features are not considered in a higher-order setting, which is a significant limitation. One problem is that features such as coroutines and locally declared exceptions are "hybrid effects" which manipulate control flow but have "state-like" features, making them difficult to capture via translation into λ_{Aff} on its own. Game semantics has been proposed as the basis for a detailed semantic account of such hybrid effects [14], since it allows state and control to be combined seamlessly. However, one apparent difficulty in giving a linear continuation passing semantics of state and state-like features is that by storing continuations, a program can perform jumps in the flow of control whilst still using its continuations linearly. In the work described here, this corresponds to the reliance on *innocence* (i.e. purely functional behaviour) to prove fullness of the embedding corresponding to linear CPS translation.

Acknowledgements

This research was supported by UK EPSRC grant GR/N 38824. I would like to thank Guy McCusker, Thomas Streicher, Peter O'Hearn, Russ Harmer and Matthew Wall for useful discussions.

References

[1] M. Abadi and M. P. Fiore. Syntactic considerations on recursive types. In *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science, LICS '96*, 1996.

[2] S. Abramsky. Axioms for full abstraction and full completeness. In *Essays in Honour of Robin Milner*. MIT Press, 1997.

[3] S. Abramsky and G. McCusker. Call-by-value games. In M. Neilsen and W. Thomas, editors, *Computer Science Logic:* 11^{th} *Annual workshop proceedings*, LNCS, pages 1–17. Springer-Verlag, 1998.

[4] S. Abramsky, R. Jagadeesan and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 2000.

[5] A. Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, LFCS, University of Edinburgh, 1996.

[6] Josh Berdine, Peter O'Hearn, Uday Reddy, and Hayo Thielecke. Linear continuation-passing. *Higher Order Symbolic Computation*, 15(2/3):181–208, September 2002.

[7] Josh Berdine, Peter W. O'Hearn, and Hayo Thielecke. On affine typing and completeness of CPS. Draft, December 2000.

[8] C. Faggian and J. M. E. Hyland. Designs disputes and strategies. In *Proccedings of CSL '02*, 2002.

[9] M. Hasegawa. Linearly used effects: monadic and cps transformations into the linear lambda calculus. In *Proc. 6th International Symposium on Functional and Logic Programming (FLOPS2002), Aizu*, number 2441 in LNCS, pages 167–182. Springer, 2002.

[10] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. *Information and Computation*, 163:285–408, 2000.

[11] J. Laird. Full abstraction for functional languages with control. In *Proceedings of the Twelfth International Symposium on Logic In Computer Science, LICS '97*. IEEE Computer Society Press, 1997.

[12] J. Laird. *A Semantic Analysis of Control*. PhD thesis, Department of Computer Science, University of Edinburgh, 1998.

[13] J. Laird. Finite models and full completeness. In *Proceedings of CSL '00*, number 1862 in LNCS. Springer, 2000.

[14] J. Laird. A fully abstract game semantics of local exceptions. In *Proceedings of the Sixteenth International Symposium on Logic In Computer Science, LICS '01*. IEEE Computer Society Press, 2001.

[15] O. Laurent. Polarized games. In *Proceedings of the Seventeenth International Symposium on Logic In Computer Science, LICS '02*, 2002.

[16] G. McCusker. *Games and full abstraction for a functional metalanguage with recursive types*. PhD thesis, Imperial College London, 1996.

[17] A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.

[18] T. Streicher. A domain equation for the designs of Ludics. 2002.

Counting and Equality Constraints for Multitree Automata

Denis Lugiez

Lab. d'Informatique Fondamentale, UMR 6166
CNRS & Université de Provence.
CMI 39 av. Joliot Curie, 13453 Marseille Cedex, France.
lugiez@cmi.univ-mrs.fr

Abstract. Multitree are unranked, unordered trees and occur in many Computer Science applications like rewriting and logic, knowledge representation, XML queries, typing for concurrent systems, cryptographic protocols.... We define constrained multitree automata which accept sets of multitrees where the constraints are expressed in a first-order theory of multisets with counting formulae which is very expressive and decidable. The resulting class of multitree automata is closed under boolean combination, has a decidable emptiness problem and we show that this class strictly embeds all previous classes of similar devices which have been defined for a whole variety of applications.

Introduction

Tree automata and regular tree languages have been used successfully in many areas of Computer Science like type systems, rewriting, program analysis, protocol verification... However the expressive power of regular languages is often too weak and many extensions of regular languages have been proposed for solving some specific problems. One trend is to add constraints to the transition rules such that a rule is used only when the constraint is satisfied. The most natural extension is to add equality constraints which state that some subterms at some positions must be equal or different. The resulting class has good closure properties but the emptiness problem (decide if the language $\mathcal{L}(\mathcal{A})$ accepted by an automaton $\mathcal{A}$ is empty or not?) is undecidable. Therefore equality constraints have been restricted to get classes with a decidable emptiness problem: automata with equality/disequality constraints between brothers [BT92], reduction automata [CCC$^+$94], and automata with a bounded number of equality test along an accepting run [CJ94]. These automata are used in automated theorem proving and rewriting. Another direction for extending the expressivity of regular languages is to use axioms, especially associativity and associativity-commutativity axioms. Hedge automata [PQ68, Mur01] which are used in query languages for XML have rules where the left-hand side is a regular expression on the set of states. Automata where the left-hand side is a Presburger formula or a rational expression of vectors of integers have be proposed for applications in

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 328–342, 2003.

type system used in verification of infinite-state systems [Col02], inductive theorem proving [LM94] or knowledge representation [NP93] for which feature tree automata are introduced. In [Ohs01], one adds equality steps involving axioms during the acceptance process.

The next step is to combine both constraints and axioms which gives two possible reasons for getting into trouble. Since the most frequently used axiom is associativity-commutativity and the most frequently needed constraints are equality constraints, it is natural to look for tree automata combining these features. The underlying data structure is now the multitree structure where some operators have unbounded arity and the subterms are unordered. Therefore the first point is to define a theory of constraints which extends equality to multitrees and possibly adds some new constraints. The second point is to define a class of multitree automata which uses the constraints and still has the good properties required by applications (basically closure under boolean combination and decision of emptiness). We present a new class of multitree automata with a simple and natural definition, where the constraints are formulae of the first-order theory of equality for multisets enriched by Presburger constraints on the cardinality of multisets. For instance, one may constraint a rule $f(q, q, q) \to q'$ by a formula saying that the first subterm reaching q has twice as elements as the second subterm reaching q and that the first subterm is the union of the second subterm and of the third one. We show that these constraints are decidable (section 2), then we prove that the multitree automata class is closed under boolean combinations (section 4), and has an elementary emptiness problem in section 5. Then we prove that this class contains all known classes of tree automata which use AC axioms or/and equality constraints, are closed under boolean combinations and have a decidable emptiness problem (in section 6). Missing proofs will appear in the long version of the paper.

In this paper we focus on the definition of the new class and its basic properties. Since this class contains many classes previously known which have been extensively used in Computer Science, we can hint at many applications already known in knowledge representation (feature tree automata and feature logic), typing for infinite state systems (rational tree automata), inductive theorem proving (automata with Presburger constraint), logic and rewriting(automata with equality constraint between brothers)... Applications to cryptographic protocols in the spirit of [GLV02] which uses two-way automata with associativity-commutativity is also relevant.

1 Notations

Terms and Multitrees. Multisets on a (finite or infinite) set of elements $e_1, e_2, \ldots$ are sets where elements can be repeated. The empty multiset is denoted by $\emptyset$. The multiset composed of $e_{i_1}, \ldots, e_{i_p}$ (where one may have $e_{i_l} = e_{i_j}$) is denoted by $\{e_{i_1}, \ldots, e_{i_p}\}$. The number of repetition of an element is its multiplicity. The number of elements of a multiset $\#_E(M)$ is the number of elements counted

with their multiplicities and $\#_D(M)$ denotes the number of distinct elements. For instance $\#_D(\{e_1, e_1, e_2\}) = 2$ and $\#_E(\{e_1, e_1, e_2\}) = 3$.

We consider terms on a finite set of free function symbols F and a finite set of binary function symbols $\oplus_1, \oplus_2, \ldots$ which are supposed to be associative-commutative (AC in short). For simplicity we use only one $\oplus$ operator, but our results are extended easily to the case of several AC symbols. The set of terms is $T_{\mathcal{F}}$ for $\mathcal{F} = F \cup \{\oplus\}$. Terms can be flattened using the rules $(x \oplus y) \oplus z) \rightarrow$ and writting a term $((\ldots(t_1 \oplus t_2) \oplus \ldots) \oplus t_n)$ as $t_1 \oplus t_2 \ldots \oplus t_n$. *Multitrees* corresponds to flattened terms but where the $\oplus$ operator is considered as a multiset constructor and are described by the grammar:

$$\begin{array}{lll} \mathcal{MT} ::= \mathcal{S} \mid \mathcal{T} & & \\ \mathcal{S} \quad ::= \mathcal{T}_1 \oplus \ldots \oplus \mathcal{T}_n & n \geq 1 & (\textit{sum–like multitrees}) \\ \mathcal{T} \quad ::= f(\mathcal{MT}_1, \ldots, \mathcal{MT}_n) & \textit{arity}(f) = n & (\textit{term–like multitrees}) \end{array}$$

For instance $f(a \oplus g(b), a \oplus a) \in \mathcal{T}$, $a \oplus a \oplus f(a, a) \in \mathcal{S}$. In the following we may say terms as well as multitrees for elements of $\mathcal{MT}$. A multitree $t_1 \oplus \ldots \oplus t_n$ is often denoted by $\Sigma_{i=1,\ldots,n} t_i$. Multitrees are equal up to permutation of arguments of $\oplus$. For instance, $f(a \oplus g(b), a \oplus b) = f(g(b) \oplus a, b \oplus a)$.

This paper deals with automata recognizing sets of multitrees.

Presburger Arithmetic. Let $\mathbb{N}$ be the set of natural numbers and let $+$ denote addition of natural numbers. Then the first-order theory of equality on this structure is called Presburger arithmetic and is decidable [1]. Diophantine equations, inequations are example of Presburger arithmetic formula (with a lower complexity since they are in NP). The models of Presburger arithmetic formulae are the *semilinear sets*.

2 First-Order Theory of Multisets with Cardinality Constraints

We define $FO_{\#}(\mathcal{M})$ the first-order theory of multisets with cardinality constraints.

The syntax of formula. Let $\mathcal{X} = \{X, Y, \ldots\}$ be a set of multiset variables, the set of terms is defined by the grammar:

$$T ::= X \mid T \oplus T$$

where the $\oplus$ operator is a binary associative-commutative symbol. We use also two unary symbols $\#_D$ and $\#_E$. The predicate is the equality predicate $=$. We also assume that $N_1, N_2, \ldots$ is a denumerable set of integer variables. Formula are given according to the grammar:

$$\phi ::= \ (T = T) \mid \psi(\#(X_1), \ldots, \#(X_n), N_1, \ldots, N_p) \mid \neg\phi \mid \phi \wedge \phi \mid \exists X \ \phi \mid \exists N \ \phi$$

where ψ is a Presburger arithmetic formula, $\#$ denotes $\#_D$ or $\#_E$.

[1] it is ATIME(double-expo,poly)-complete [Ber77]

Semantics. Let $\mathcal{M}$ be the set of finite multisets built on a denumerable set of distinct elements $e_1, e_2, \ldots$. An interpretation I associates to each variable X a multiset $I(X) \in \mathcal{M}$, and to each integer variable N some natural number $I(N) \in \mathbb{N}$. The function $\oplus$ is interpreted as the union of multisets, equality is equality of multisets, and $\#_D(X)$ (resp. $\#_E$) is interpreted as the number of distinct elements (resp. number of elements) of X. The interpretation is extended to formula as in first-order logic, and similarly we define satisfiability, models,... The values of the e_i's are not relevant for the satisfiability of a formula and satisfiability is preserved by one-to-one mapping of the e_i's.

Expressivity. This logic can express many natural properties.

- any Presburger formula related to the number of elements of multisets (counting formula). For instance X has as many elements as Y: $\#_E(X) = \#_E(Y)$.
- X is empty: $\#_E(X) = 0$, $\quad X$ is a singleton: $\#_E(X) = 1$
- Y is a subset of X: $\exists Z : X = Y \oplus Z$
- the intersection $X \cap Y$ is empty: $\forall Z : Z \subseteq X \wedge Z \subseteq Y \Rightarrow \mathit{empty}(Z)$
- X is a multiset containing only copies of some element: $\#_D(X) = 1$
- X is a multiset s.t. the multiplicity n_e of each element e satisfies the Presburger formula $\psi(n_e)$:

$$\forall X_e \; Y \; X = X_e \oplus Y \wedge \#_D(X_e) = 1 \wedge X_e \cap Y = \emptyset \Rightarrow \psi(\#_E(X_e))$$

  This formula is called $Mult(\psi, X)$. This is extended for a tuple $X_1, \ldots, X_n$ and a variable ψ with n free variables, yielding a formula $Mult(\psi, X_1, \ldots, X_n)$.
- N is the maximal multiplicity of an element of X:

$$\begin{array}{l} \forall X_e, Y \; X = X_e \oplus Y \wedge \#_D(X_e) = 1 \wedge X_e \cap Y = \emptyset \Rightarrow \#_E(X_e) \leq N \\ \wedge \exists X_e, Y \; X = X_e \oplus Y \wedge \#_D(X_e) = 1 \wedge X_e \cap Y = \emptyset \wedge \#_E(X_e) = N \end{array}$$

  We shall abbreviate this formula into $N = \#_M(X)$.
- Y is the set of distinct elements of X:

$$\begin{array}{r} Y \subseteq X \wedge \forall X_e, X' \; (X = X_e \oplus X' \wedge X_e \neq \emptyset \Rightarrow Y \cap X_e \neq \emptyset) \\ \wedge \forall Y_e, Y' \; (Y = Y_e \oplus Y' \Rightarrow Y_e \cap Y' = \emptyset) \end{array}$$

  then $\#_D(X) = \#_E(Y)$ which shows that $\#_D$ is definable within the logic.

The extension of $FO_{\#}(\mathcal{M})$ with variables $x, y, \ldots$ for elements and the membership predicate $x \in X$ is achieved by introducing a multiset variable X_x for each variable x together with the condition that X_x is a singleton, and replacing $x \in X$ by $X_x \subseteq X$.

Theorem 2.1. *The first-order theory of multisets with cardinality constraints is decidable.*

When there is no $\#(X)$ occurrence, the result can be obtained by encoding the multiset theory in Skolem arithmetic (private communication from Achim Blumensath). The extended version of the paper gives an alternative proof based on semilinear sets which provides an explicit representation of the model of a formula. When no multiset variable occur free, we get:

Proposition 2.1. *The model of a formula $\phi(N_1, \ldots, N_p)$ of $FO_{\#}(\mathcal{M})$ is a semi-linear set constructible in elementary time.*

3 Multitree Automata with Constraints

Equality of multitrees up to permutation of elements of $\oplus$ defines an equivalence relation. Let $e_1, e_2, \ldots$ be a enumeration of the equivalence classes corresponding to the elements of $\mathcal{T}$. For simplicity we identify an element of $\mathcal{T}$ and its equivalence class. We interpret each multitree t in $\mathcal{MT} = \mathcal{S} \cup \mathcal{T}$ as a multiset $[\![t]\!]$ of e_i's as follows: - if $t \in \mathcal{T}$ then t is in some e_i and we set $[\![t]\!] = \{e_i\}$
- if $t \in \mathcal{S}$ then $t = e_{i_1} \oplus \ldots \oplus e_{i_p}$ and $[\![t]\!] = \{e_{i_1}, \ldots, e_{i_p}\}$
For instance $[\![f(a \oplus b)]\!] = \{f(a \oplus b)\}$ and $[\![a \oplus b]\!] = \{a, b\}$.

Definition 3.1. *A multitree automaton is composed of a finite set of states $\mathcal{Q} = \{q_1, \ldots, q_m\}$, a set of final states $\mathcal{Q}_{Final} \subseteq \mathcal{Q}$ and a set of rules R of the form: (type 1) $\phi(X_1, \ldots, X_n) \Rightarrow f(q_1, \ldots, q_n) \to q$ for f of arity n*
(type 2) $\phi(X_{q_1}, \ldots, X_{q_m}) \Rightarrow q$
where in each case, ϕ denotes a formula of $FO_{\#}(\mathcal{M})$.

The transition relation $\to_{\mathcal{A}}$ is defined by $t \to_{\mathcal{A}} q$ iff

$$t = f(t_1, \ldots, t_n) \to_{\mathcal{A}} q \text{ if } \begin{array}{l} \phi(X_1, \ldots, X_n) \Rightarrow f(q_1, \ldots, q_n) \to q \in R \\ t_i \to_{\mathcal{A}} q_i \text{ for } i = 1, \ldots, n \\ \models \phi([\![t_1]\!], \ldots, [\![t_n]\!]) \end{array}$$

$$t = e_1 \oplus \ldots \oplus e_p \to_{\mathcal{A}} q \text{ if } \begin{array}{l} \phi(X_{q_1}, \ldots, X_{q_m}) \Rightarrow q \in R \\ t = t_1 \oplus \ldots \oplus t_m \text{ where, for } i = 1, \ldots, m, \\ t_i = e_{i,1} \oplus \ldots \oplus e_{i,n_i} \text{ and } e_{i,j} \to_{\mathcal{A}} q_i \text{ for } j = 1, \ldots, n_i \\ \models \phi([\![t_1]\!], \ldots, [\![t_m]\!]) \end{array}$$

A multitree is *accepted* iff $t \to_{\mathcal{A}} q$ with $q \in \mathcal{Q}_{Final}$. The language $\mathcal{L}(\mathcal{A})$ accepted by $\mathcal{A}$ is the set of multitrees accepted by $\mathcal{A}$.

Example 3.1. Given a signature consisting of two constants a, b, one binary symbol f, an automaton accepting only multisets with two constants a and b such that the number of b's is greater than the number of a's can be $\mathcal{A} = (\{q_a, q_b, q_S\}, \{q_S\}, R)$ with $R = \{True \Rightarrow a \to q_a, True \Rightarrow b \to q_b, \#_E(X_{q_a}) < \#_E(X_{q_b}) \Rightarrow q_S\ \}$.

Then $a \oplus b \oplus b \to q_S$ since $\begin{cases} a \to q_a,\ b \to q_b,\ [\![a]\!] = \{a\} \text{ and } [\![b \oplus b]\!] = \{b, b\} \\ \models \#_E([\![a]\!]) < \#_E([\![b \oplus b]\!]) \end{cases}$

To accept also the multitrees s.t. that each subterm $f(t_1, t_2)$ satisfies $t_1 \neq t_2$ and $t_1, t_2 \in \mathcal{L}(\mathcal{A})$, we simply add the rule: $X_1 \neq X_2 \Rightarrow f(q_S, q_S) \to q_S$ □

Two automata are *equivalent* if they have the same language. The class of multitree languages accepted by multitree automata with constraints is denoted by $CMTL$. For simplicity, it is easier to consider automata such that

(i) $\mathcal{Q} = \mathcal{Q}_\mathcal{T} \cup \mathcal{Q}_\mathcal{S}$ with $\mathcal{Q}_\mathcal{T} \cap \mathcal{Q}_\mathcal{S} = \emptyset$,
(ii) for all type 1 rules $\phi(X_1, \ldots, X_n) \Rightarrow f(q_1, \ldots, q_n) \to q$, we have $q \in \mathcal{Q}_\mathcal{T}$
(iii) for all type 2 rules $\phi(X_{q_1}, \ldots, X_{q_m}) \Rightarrow q$, we have $q \in \mathcal{Q}_\mathcal{S}$
and $\phi(X_{q_1}, \ldots, X_{q_m}) \equiv \phi'(X_{q_1}, \ldots, X_{q_{m'}})$ where $\{q_1, \ldots, q_{m'}\} = \mathcal{Q}_\mathcal{T}$.

This can be achieved by splitting each state q into q_M and q_F, replacing type 1 rules $\ldots \to q$ by $\ldots \to q_F$ and type 2 rules $\ldots \Rightarrow q$ by $\ldots \Rightarrow q_M$ and any occurrence of q elsewhere by q_M and q_F.

Example 3.2. In the second automaton of the previous example, the state q_S can be reached by multitrees of $\mathcal{S}$ as well as multitrees of $\mathcal{T}$. Therefore we split it into $q_S^\mathcal{S}$ and $q_S^\mathcal{T}$ and replace the rule $X_1 \neq X_2 \Rightarrow f(q_S, q_S) \to q_S$ by the rules $X_1 \neq X_2 \Rightarrow f(\_, \_) \to q_S^\mathcal{T}$ where $\_$ is any of $q_S^\mathcal{T}, q_S^\mathcal{T}$, and the rule $\#_E(X_{q_a}) < \#_E(X_{q_b}) \Rightarrow q_S$ by $\#_E(X_{q_a}) < \#_E(X_{q_b}) \Rightarrow q_S^\mathcal{S}$. □

4 Properties of Multitree Automata with Constraints

Membership. Given an automaton $\mathcal{A}$, its size $|\mathcal{A}|$ is the number of symbols of its presentation, $C(t)$ is a bound on the time for checking the satisfiability of constraints of $\mathcal{A}$ on the subterms of t. If the constraints are quantifier-free formulas, this amounts to solving equality of terms modulo AC which can be solved in polynomial time in $|t|$, see [BKN85].

Proposition 4.1. *$t \in \mathcal{L}(\mathcal{A})$ is decidable in time $O(|\mathcal{A}||t||C(t)|)$*

Proof. Consider all $|\mathcal{A}||t|$ possible labelling of nodes in t where the root is labelled by a final state, and check the applicability of rules. □

Completion. An automaton is *complete* if each multitree reaches at least one state. To get a complete automaton equivalent to a given automaton, we add a sink state q_S, the rules $True \Rightarrow f(\ldots, q_S, \ldots) \to q_S$ and the rules $True \Rightarrow q_S$.

Determinization. An automaton is *deterministic* iff for each multitree t, there exists at most one state q such that $t \to_\mathcal{A} q$. We show how to build a deterministic automaton $\mathcal{A}_D$ equivalent to a non-deterministic automaton $\mathcal{A} = (\mathcal{Q}_\mathcal{A}, \mathcal{Q}_{Final}, R)$. The deterministic automaton $\mathcal{A}_D$ has a set of states $\mathcal{Q}_D = 2^{\mathcal{Q}_\mathcal{A}}$, the final states are the states containing a final state of $\mathcal{A}$.

Determinization of conditions. First we replace rules by rules such that constraints are pairwise incompatible. For a symbol f of arity n, let $\phi_i(X_1, \ldots, X_n)$ for $i = 1, \ldots, m$ be the conditions of corresponding type 1 rules. For each $I \subseteq \{1, \ldots, m\}$, let $\psi_I(X_1, \ldots, X_n)$ be defined by

$$\bigwedge_{i \in I} \phi_i(X_1, \ldots, X_n) \wedge \bigwedge_{i \notin I} \neg \phi_i(X_1, \ldots, X_n)$$

By construction $\psi_I(X_1, \ldots, X_n) \wedge \psi_J(X_1, \ldots, X_n)$ is unsatisfiable if $I \neq J$ and $\phi_i(X_1, \ldots, X_n) \Leftrightarrow \bigvee_{i \in I} \psi_I(X_1, \ldots, X_n)$. Then each rule $\phi_i(X_1, \ldots, X_n) \Rightarrow f(q_1, \ldots, q_n) \to q$ is replaced by the rules $\psi_I(X_1, \ldots, X_n) \Rightarrow f(q_1, \ldots, q_n) \to q$ for $i \in I$.

The subset construction: type 1 rules. Type 1 rules of $\mathcal{A}_D$ are

$$\psi_I(X_1,\ldots,X_n) : f(Q_1,\ldots,Q_n) \to Q$$

with $Q = \{q \mid \exists q_1 \in Q_1,\ldots,q_n \in Q_n,\ \psi_I(X_1,\ldots,X_n) \Rightarrow f(q_1,\ldots,q_n) \to q \in R\}$

The subset construction: type 2 rules. For $J \subseteq \{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}$, let Q_J denote $\{q_j \mid j \in J\}$ and let X_J the variable associated to Q_J in type 2 rules of $\mathcal{A}_D$. Let the type 2 rules of $\mathcal{A}$ be $\phi_1(X_1,\ldots,X_{|\mathcal{Q}_\mathcal{A}|}) \Rightarrow q_1,\ \ldots,\ \phi_p(X_1,\ldots,X_{|\mathcal{Q}_\mathcal{A}|}) \Rightarrow q_p$ For each $k = 1,\ldots,p$, we define $\psi_k(X_\emptyset,\ldots,X_J,\ldots,X_{\{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}})$ by:

$$\bigwedge_{J\subseteq\{1,..,|\mathcal{Q}_\mathcal{A}|\}} (\exists X_j^J\ X_J = \Sigma_{j\in J}X_j^J \wedge \phi_k(\Sigma_{J\subseteq\{1,..,|\mathcal{Q}_\mathcal{A}|\}}X_1^J,..,\Sigma_{J\subseteq\{1,..,|\mathcal{Q}_\mathcal{A}|\}}X_{|\mathcal{Q}_\mathcal{A}|}^J)$$

The idea underlying the construction of ψ_k is the following one: X_J represents a sum of e_i's s.t. each e_i reaches exactly all the states q_j for $j \in J$. In a derivation of $\mathcal{A}$, each e_i reaches only one of the possible q_j which is represented by the decomposition of X_J into the sum $\Sigma_{j\in J}X_j^J$. Finally, we sum all terms that reach the same state $q_i \in Q_\mathcal{A}$ for $i = 1,\ldots,|\mathcal{Q}_\mathcal{A}|$, and we check whether the condition ϕ_k is satisfiable, which means that the state q_k can be reached.

The last point is to eliminate the remaining ambiguities (since the same term can satisfy several ψ_k formulas) yielding the following type 2 rules of $\mathcal{A}_D$:

$$\bigwedge_{i\in I}\psi_i(X_\emptyset,\ldots,X_{\{1,\ldots,\mathcal{Q}_\mathcal{A}\}}) \wedge \bigwedge_{i\notin I}\neg\psi_i(X_\emptyset,\ldots,X_{\{1,\ldots,\mathcal{Q}_\mathcal{A}\}}) \Rightarrow Q_I$$

Proposition 4.2. *$|\mathcal{A}_D| = O(2^{2|\mathcal{A}|})$ and $t \to_{\mathcal{A}_D} Q$ iff $Q = \{q \mid t \to_\mathcal{A} q\}$.*

Proof. We show that $t \to_{\mathcal{A}_D} Q$ iff $Q = \{q \mid t \to_\mathcal{A} q\}$ by structural induction on t.

Case $t = f(t_1,\ldots,t_n)$. Since the conditions of rules are either identical or pairwise incompatible, the proof is similar to the correctness proof for the determinization of tree automata.

Case $t = t_1 \oplus \ldots \oplus t_n$. Assume that the property holds for the t_i's. We denote by $t \to_\mathcal{A} I$ the property that $I = \{i \mid t \to_\mathcal{A} q_i\}$ for any multitree t. We can write

$$t = \Sigma_{J\subseteq\{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}}\Sigma_{t_j\to_\mathcal{A} J}t_j = \Sigma_{J\subseteq\{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}}T_J \text{ with } T_J = \Sigma_{t_j\to_\mathcal{A} J}t_j$$

where the decomposition is unique by induction hypothesis.

- Assume that $t \to_{\mathcal{A}_D} Q_I$ using $\bigwedge_{i\in I}\psi_i(X_\emptyset,\ldots,X_{\{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}}) \wedge \bigwedge_{i\notin I}\neg\psi_i(X_\emptyset,\ldots,X_{\{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}}) \Rightarrow Q_I$. Let $i \in I$. By definition $\models \psi_i(T_\emptyset,\ldots,T_{\{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}})$ Therefore, for all $J \subseteq \{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}$ we find a decomposition[2] $T_J = \Sigma_{j\in J}T_j^J$ such that

$$\models \phi_i(\Sigma_{J\subseteq\{1,\ldots\mathcal{Q}_\mathcal{A}\}}T_1^J,\ldots,\Sigma_{J\subseteq\{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}}T_{|\mathcal{Q}_\mathcal{A}|}^J)$$

[2] this decomposition depends on i but we don't write this explicitely for simplicity

This proves that $t = \Sigma_{j=1}^{j=|\mathcal{Q}_\mathcal{A}|} \Sigma_{J\subseteq\{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}} T_j^J \rightarrow_\mathcal{A} q_i$.

For $i \notin I$, we can't find any such decomposition by definition of $\neg\psi_i$ (otherwise $t \models \psi_i$ for $i \notin I$ and $t \not\rightarrow_{\mathcal{A}_D} Q_I$), which proves that $t \not\rightarrow_\mathcal{A} q_i$.
Combining the two previous results, we get $Q_I = \{q_i \mid t \rightarrow_\mathcal{A} q_i\}$.

– Conversely, let $Q = \{q \mid t \rightarrow_\mathcal{A} q\} = Q_I$ for some I. According to our notation,

$$t = \Sigma_{J\subseteq\{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}} T_J \text{ where } T_J = \Sigma_{t_j \rightarrow_\mathcal{A} J} t_j$$

By definition of Q_I, for $i \in I$, there is a decomposition[3] $T_J = \Sigma_{j\in J} t_j^J$ s.t.

$$\phi_i(\Sigma_{J\subseteq\{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}} t_1^J, \ldots, \Sigma_{J\subseteq\{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}} t_{|\mathcal{Q}_\mathcal{A}|}^J)$$

This proves that $\models \psi_i(T_\emptyset, \ldots, T_{\{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}})$.

For $i \notin I$ there is no such decomposition (otherwise $t \rightarrow_\mathcal{A} q_i$), therefore $\not\models \psi_i(T_\emptyset, \ldots, T_{\{1,\ldots,|\mathcal{Q}_\mathcal{A}|\}})$.
Combining the two properties we get that $t \rightarrow_{\mathcal{A}_D} Q_I$ □

Compositional properties: Product, union, intersection. Given $\mathcal{A} = (\mathcal{Q} = \{q_1, \ldots, q_n\}, \mathcal{Q}_{Final}, R)$, $\mathcal{A}' = (\mathcal{Q}'\{q'_1, \ldots, q'_{n'}\}, \mathcal{Q}'_{Final}, R')$ two automata, the set of states of the product $\mathcal{A} \times \mathcal{A}'$ is $\mathcal{Q}_\times = \mathcal{Q} \times \mathcal{Q}'$, the set of final states is empty, and the rules are given by:

(type 1) $\phi(X_1, \ldots, X_n) \wedge \phi'(X_1, \ldots, X_n) \Rightarrow f((q_1, q'_1), \ldots, (q_n, q'_n)) \rightarrow (q, q')$
iff $\begin{cases} \phi(X_1, \ldots, X_n) \Rightarrow f(q_1, \ldots, q_n) \rightarrow q \in R, \\ \phi'(X_1, \ldots, X_n) \Rightarrow f(q'_1, \ldots, q'_n) \rightarrow q' \in R' \end{cases}$

(type 2) $\phi(\Sigma_{j\in\{1,\ldots,n'\}} X_{(q_1,q'_j)}, \ldots, \Sigma_{j\in\{1,\ldots,n'\}} X_{(q_n,q'_j)}) \Rightarrow (q, q')$
$\wedge\ \phi'(\Sigma_{i\in\{1,\ldots,n\}} X_{(q_i,q'_1)}, \ldots, \Sigma_{i\in\{1,\ldots,n\}} X_{(q_i,q'_{n'})})$
iff $\begin{cases} \phi(X_{q_1}, \ldots, X_{q_n}) \Rightarrow q \in R \\ \phi'(X_{q'_1}, \ldots, X_{q'_{n'}}) \Rightarrow q' \in R' \end{cases}$

Proposition 4.3. *The construction of $\mathcal{A} \times \mathcal{A}'$ is done in time $O(|\mathcal{A}||\mathcal{A}'|)$ and $t \rightarrow_{\mathcal{A}\times\mathcal{A}'} (q, q')$ iff $t \rightarrow_\mathcal{A} q$ and $t \rightarrow_{\mathcal{A}'} q'$*

From this proposition we get closure under intersection and union (simply adjust the set of final states accordingly).

Complementation Complementation is straightforward for a complete deterministic automata: exchange final and non-final states. Since every automaton is equivalent to a complete deterministic one, we are done.

[3] again, we don't mention explicitely that the decomposition depends on i

5 Decision of Emptiness

Now we come to our most technical result. The principle of the algorithm for deciding emptiness of $\mathcal{L}(\mathcal{A})$ is the same as for all classes of (finite) tree automata: it is a marking algorithm which marks all reachable states until no new state can be marked. As usual, constraints make life more difficult: given a rule $X_1 \neq X_2 \Rightarrow f(q,q) \rightarrow q'$, we can't mark the state q as soon as we know that some multitree reach this state since the satisfiability of the constraint $X_1 \neq X_2$ requires that at least two different multitrees reach q. Actually this ensures that also two multitrees reach q', establishing an invariant property of the marking algorithm. Since we deal with multitrees, we shall use two bounds: D on the number of different multitrees reaching each state, and M on the maximal multiplicity of an element in a multitree. These bounds are computed from the constraints of the rules and they are effectively computable because $FO_{\#}(\mathcal{M})$ is decidable.

Remark 5.1. Emptiness can be decided directly for a non-deterministic automaton, but in this case the algorithm realizes an implicit determinization which complicates the construction without adding significant improvments. Therefore we shall give the algorithm for deciding the emptiness of the language accepted by a deterministic automaton.

5.1 Formulae for States

Let $\mathcal{A} = (\mathcal{Q}, \mathcal{Q}_{Final}, R)$ be a deterministic automaton. We assume that $\mathcal{Q}$ is the disjoint union of $\mathcal{Q}_{\mathcal{S}}$ and $\mathcal{Q}_{\mathcal{T}} = \{q_1, \ldots, q_p\}$ such that only multitrees of $\mathcal{T}$ can reach a state of $\mathcal{Q}_{\mathcal{T}}$ and only multitrees of $\mathcal{S}$ can reach a state of $\mathcal{Q}_{\mathcal{S}}$. We now write formulae which ensure that a state q can be reached by some multitree X, when we have already computed Z_1 a set of multitrees of $\mathcal{T}$ reaching q_1,..., Z_p a set of multitrees of $\mathcal{T}$ reaching q_p. Since the automaton is deterministic, we have $Z_i \cap Z_j = \emptyset$ if $i \neq j$. We use the notation $set(X)$ for the multiset equal to the set of distinct elements of X (this can be defined in $FO_{\#}(\mathcal{M})$, see section 2).

Case of a state $q \in \mathcal{Q}_{\mathcal{S}}$. Let $\phi_i(X_{q_1}, \ldots, X_{q_p}) \Rightarrow q$ for $i = 1, \ldots, l$ be the type 2 rules for q. The formula $\psi_q(Z_1, \ldots, Z_p, X)$ states that X reaches q when Z_1 is a set of multitrees reaching q_1,...,Z_p a set of multitrees reaching q_p and it is defined by:

$$\bigvee_{i=1}^{i=l} (\exists X^i_{q_1}, \ldots, X^i_{q_p} \; X = \Sigma_{j=1}^{j=p} X^i_{q_j} \wedge \bigwedge_{j=1}^{j=p} set(X^i_{q_j}) \subseteq Z_j \wedge \phi_i(X^i_{q_1}, \ldots, X^i_{q_p}))$$
$$/* \; there \; is \; some \; rule \; reaching \; q \; that \; can \; be \; fired \; for \; X \; */$$

Case of a state $q \in \mathcal{Q}_{\mathcal{T}}$. First, we define the formula $X \in L_q$ which expresses that the multitree X is in the language accepted by q by:

- $\#(X) = 1 \wedge X \subseteq Z_i$ if q is some $q_i \in \mathcal{Q}_{\mathcal{T}}$
- $\psi_q(Z_1, \ldots, Z_p, X)$ if $q \in \mathcal{Q}_{\mathcal{S}}$

Note that the definition is consistent since ψ_q is already defined for $q \in \mathcal{Q}_\mathcal{S}$. For simplicity we assume that $\phi_i(X_{q_1^i}, \ldots, X_{q_n^i}) \Rightarrow f(q_1^i, \ldots, q_n^i) \rightarrow q$ for $i = 1, \ldots, l$ are the type 2 rules for q (this can be achieved easily modulo introducing new states). The formula $\psi_q((Z_1, \ldots, Z_p, X_1, \ldots, X_n)$ for $q \in \mathcal{Q}_\mathcal{T}$ is defined by:

$$\bigvee_{i=1}^{i=l}(\bigwedge_{j=1}^{j=n} X_j \in L(q_j^i) \wedge \phi_i(X_1, \ldots, X_n))$$
$$/* \textit{ there is some rule reaching } q \textit{ that can be fired for } f(X_1, \ldots, X_n) */$$

5.2 Minimal Solutions of Formulae in $FO_\#(\mathcal{M})$

Let $\phi(N_1, \ldots, N_p)$ be a formula of $FO_\#(\mathcal{M})$ with no multiset free variables, a p-uple $(n_1, \ldots, n_p) \in \mathbb{N}^p$ is *minimal* for ϕ iff (i) $\models \phi(n_1, \ldots, n_p)$ and (ii) there is no $(n'_1, \ldots, n'_p)$ s.t. $\models \phi(n'_1, \ldots, n'_p) \wedge \bigwedge_{i=1}^{i=p} n'_i < n_i$. Conditions (i) and (ii) are expressible by a formula $Minimal_\phi(N_1, \ldots, N_p)$ of $FO_\#(\mathcal{M})$. The set of p-uples minimal for ϕ is a semilinear set computable in elementary time (by proposition 2.1). The *minimum* of $\phi(N_1, \ldots, N_p)$, denoted by $m = Min(\phi)$ is the unique m minimal for $\psi(M) \equiv \exists N_1, \ldots, N_p \; Minimal_\phi(N_1, \ldots, N_p) \wedge \bigwedge_{i=1}^{i=p} M \geq N_i$. When the formula is unsatisfiable we set $m = +\infty$. By definition m is the smallest natural number which is greater that any component of any minimal solution of ϕ (if there exists one) and m is computable in elementary time.

5.3 Bounds for States

We define a bound D on the number of distinct elements and a bound M on the multiplicities of elements in multitrees of $\mathcal{MT}$ that reach a state q. For a state $q \in \mathcal{Q}_\mathcal{S}$ we compute D_q as

$$Min(\exists Z_1, \ldots, Z_p, X \; \psi_q(Z_1, \ldots, Z_p, X) \wedge \bigwedge_{i=1}^{i=p} \#(Z_i) = M_i)$$

For a state $q \in \mathcal{Q}_\mathcal{T}$ associated to f of arity n we compute D_q as

$$Min(\exists Z_1, \ldots, Z_p, X_1, \ldots, X_n \; \psi_q(Z_1, \ldots, Z_p, X_1, \ldots, X_n) \wedge \bigwedge_{i=1}^{i=p} \#(Z_i) = M_i)$$

and we set D as the maximum of the finite D_q's. This value bounds the number of multitrees that must reach q_1,...,q_p to allow the construction of a multitree reaching q (if such a multitree exists). Now we compute bounds on the multiplicities of elements of Z_i used in X or $X_1, \ldots, X_n$, with the additional constraints that (i) $Z_1, \ldots, Z_p$ have less than $D+1$ elements and (ii) we can construct $D+1$ multitrees reaching q (instead of a single one). We recall that $\#_M(X)$ is the formula defining the maximal multiplicity of elements of X.

For a state $q \in \mathcal{Q}_\mathcal{S}$, for $k = 1, \ldots, D+1$ the formula $\rho_q^k(Z_1, \ldots, Z_p, X_1, \ldots, X_k)$ states that we can compute k distinct multitrees reaching q from $Z_1, \ldots, Z_p$. It is defined by

$$\bigwedge_{j=1}^{j=k} \psi_q(Z_1,\ldots,Z_p,X_j) \wedge \bigwedge_{\substack{1\leq i,j\leq k\\ i\neq j}} X_i \neq X_j$$

and we compute M_q^k as

$$Min(\exists Z_1,..,Z_p,X_1,..,X_k\ \rho_q^k(Z_1,..,Z_p,X) \wedge \bigwedge_{i=1}^{i=p} \#(Z_i) \leq D \wedge \bigwedge_{i=1}^{i=k} \#_M(X_i) \leq M)$$

which gives the bound on the multiplicities of occurrences of elements of Z_i's occurring in the X_j's for $j = 1,\ldots,k$ when we have the additionnal constraint that the number of elements of each Z_i is bounded by D.

Now, we must perform a similar computation for states of $\mathcal{Q}_\mathcal{T}$. The notation $(X_1^i,\ldots,X_n^i) \neq (X_1^i,\ldots,X_n^i)$ denotes the formula $\bigvee_{l=1}^{l=n} X_l^i \neq X_l^j$ and states that the two n-uples of multisets are distinct. For a state $q \in \mathcal{Q}_\mathcal{T}$ associated to f of arity n, for $k = 1,\ldots,D+1$, we define $\rho_q^k(Z_1,\ldots,Z_p,\underbrace{X_1^1,\ldots,X_n^1}_{first\ n-uple},\ldots,\underbrace{X_1^k,\ldots,X_n^k}_{kth\ n-uple})$ which states that we can compute k distinct multitrees $f(X_1^1,\ldots,X_n^1),\ldots,$ $f(X_1^k,\ldots,X_n^k)$ reaching q from $Z_1,\ldots,Z_p$, by

$$\bigwedge_{j=1}^{j=k} \psi_q(Z_1,\ldots,Z_p,X_1^j,\ldots,X_n^j) \wedge \bigwedge_{\substack{1\leq i,j\leq k\\ i\neq j}} (X_1^i,\ldots,X_n^i) \neq (X_1^j,\ldots,X_n^j)$$

and we compute M_q^k as

$$Min(\exists Z_1,..,Z_p,X_1^1,..,X_n^1,..,X_1^k,..,X_n^k\ \ \rho_q^k(Z_1,..,Z_p,X_1^1,..,X_n^1,..,X_1^k,..,X_n^k) \wedge \textstyle\bigwedge_{i=1}^{i=p} \#(Z_i) \leq D \wedge \bigwedge_{i=1}^{i=p} \#_M(X_i) \leq M)$$

Let M be the maximum of the finite M_q^k for $q \in \mathcal{Q}_\mathcal{S} \cup \mathcal{Q}_\mathcal{T}$ and $k = 1,\ldots,D+1$.

5.4 The Algorithm

Let D and M be as defined previously and let $\mathcal{Q} = \{q_1,\ldots,q_p\}$ be the set of states q of $\mathcal{A}$. For each $q_i \in \mathcal{Q}$, the *Reachability* algorithm computes the set $\mathcal{L}_i^m$ which is (an approximation of) the set of multitrees that reach the state q_i in m steps at most [4], where the approximation amounts to bounding the number of multitrees in $(\mathcal{L}_i^m)$ by D and the multiplicity of elements in a sum by M.

[4] One step doesn't mean one application of rule, but *label the root of a multitree by some state when all the sons are labelled by a state*

The Reachability algorithm

/*Initialize*/
$m = 0$, $\mathcal{L}_i^m = \emptyset$, set q_i unmarked for all $i = 1, \ldots, p$
/*Loop*/
repeat /*Compute the increasing sequence $(\mathcal{L}_i^m)$*/
  for all $i = 1, \ldots, p$ **do**
    if q_i is marked **then** $\mathcal{L}_i^{m+1} = \mathcal{L}_i^m$
    else $\mathcal{L}_i^{m+1} = \mathcal{L}_i^m \cup \{t \mid t \to q_i$ for $t = f(t_1, \ldots, t_n)$ with $t_i \in \mathcal{L}_j^m$
      or $t = \Sigma_j t_j$ with $t_j \in \mathcal{L}_j^m$, $\#_D(t) \leq D$
      and $\#_M(t) \leq M \quad \}$
    if $|\mathcal{L}_i^{m+1}| \geq D + 1$ **then** mark q_i
until $\mathcal{L}_i^m = \mathcal{L}_i^{m+1}$ for all $i = 1, \ldots, p$.
for each $i = 1, \ldots, p$ **do** set $\mathcal{L}_i = L_i^m$

Proposition 5.1. *The algorithm terminates and q_i is reachable iff $\mathcal{L}_i \neq \emptyset$.*

Proof. (Idea). Termination is obvious. To prove correctness, we set $L_i^0 = \emptyset$ and $L_i^{m+1} = L_i^m \cup \{t \mid t \to q_i$ *for* $t = f(t_1, \ldots, t_n)$ *and* $t_i \in L_i^m, i = 1, \ldots, n$
or
$t = t_1 \oplus \ldots \oplus t_l$ *and* $t_i \in L_{j_i}^m, i = 1, \ldots, l\}$
Then we prove that: $\forall m, i, \mathcal{L}_i^m \subseteq L_i^m$ *and* $(\mathcal{L}_i^m = L_i^m$ *or* $|\mathcal{L}_i^m| > D)$ □

An immediate consequence of the last proposition is:

Proposition 5.2. $\mathcal{L}(\mathcal{A}) = \emptyset$ *is decidable.*

The determinization process, the computations of bounds and the reachability algorithm involve only a fixed number of exponential steps. Therefore the decision procedure for emptiness is elementary.

6 Comparison with Other Classes of Tree Languages

Language with Equality/Disequality Constraints between Brothers. Tree automata with equality/disequality constraint between brothers are the most significant extension of tree automata which retains the good properties of tree automata: closure under boolean properties and decision of emptiness. This class, denoted by $L(AWEDC)$, has been used to get or improve decision results of many problems (mainly in rewriting, constraint solving and logic). No AC symbols occur in the signature and the only constraints rules have the form $\bigwedge_{i,j} X_i = X_j \wedge \bigwedge_{k,l} X_k \neq X_l \Rightarrow f(q_1, \ldots, q_n) \to q$ (a variable X_m representing the m^{th} son of the term on which the rule is tested). Such rules are a subcase of type 1 rules when no AC symbol occur, therefore these languages are particular instances of constrained multitree languages.

Proposition 6.1. $L(AWEDC) \subseteq CMTL$

The class Reg of regular languages is a subclass of $L(AWEDC)$, therefore $Reg \subset CMTL$. Unrestricted equality constraints leads to classes with an undecidable emptiness problem, but special inequality/disequality constraints have been studied leading to reduction automata [CCC+94] or automata allowing only a bounded number of equality tests in a run [CJ94]. Such constraints are different in nature to $FO_{\#}(\mathcal{M})$ constraints and can't be combined with them.

Closure of Regular Tree Languages. Regular tree languages are usually not closed under associativity or associativity-commutativity but $CMTL$ languages are closed under associativity- commutativity. Therefore the relevant question is whether the closure of a regular-tree language under AC is necessarily in $CMTL$? Let $Cl(Reg)$ denote the closure of regular languages under AC where we assume that terms are flattened i.e. transformed into multitrees, see [BN98]. The expressivity of type 2 rules allows to get the following inclusion:

Proposition 6.2. $Cl(Reg) \subseteq CMTL$

Tree Language with Rational Constraints. Tree automata with rational constraints, $TARC$ in short, have usual tree automata rules as type 1 rule and the constraints for type 2 rules are Presburger formula $\psi(\#(X_1), \ldots, \#(X_n))$. In [Col02], these constraints are rational expressions that the representation of $(\#(X_1), \ldots, \#(X_n))$ in some basis satisfies [5]. Therefore we get:

Proposition 6.3. $L(TARC) \subseteq CMTL$

Since equational tree automata defined in [Ohs01] coincide with $TARC$ (unpublished result) we get another inclusion for free.

Multitree Automata with Arithmetic Constraints. Tree automata with arithmetic constraints [LM94] work on normalized multitrees where all occurrences of the same element e are replaced by a pair (multiplicity of e, e). A normalized multitree can be denoted by $n_1.e_1 \oplus \ldots \oplus n_p.e_p$. This normalization process is costly and can't be reversed. The states of these automata are divided in several sorts that we simplify into unprimed, primed, double primed states. The relevant rules of the automata are $\phi(N) : N.q \to q'$ and $\psi(\#(q'_1), \ldots, \#(q'_m)) \to q''$ where ψ and ϕ are Presburger formula, and $\#(q)$ denotes the number of occurrences of q. Furthermore, there is no constrained rules similar to type 1 rule. Normalized multisets accepted by these automata have the form $\{\underbrace{n^1_1.e^1_1, \ldots, n^1_{k_1}.e^1_{k_p}}_{\models \phi_1(n^1_i)}, \ldots, \underbrace{n^m_1.e^m_1, \ldots, n^m_{k_m}.e^m_{k_m}}_{\models \phi_m(n^m_i)}\}$ where $\models \psi(k_1, \ldots, k_m)$ for some Presburger formula ψ. The expressivity of $FO_{\#}(\mathcal{M})$ allows to express that some (not normalized) multiset has the above form after normalization. The constraint is

$$X = X_{q'_1} \oplus \ldots \oplus X_{q'_m} \wedge \bigwedge_{i=1,\ldots,m} Mult(\phi_i, X_{q'_i}) \wedge \psi(\#_D(X_{q'_1}), \ldots, \#_D(X_{q'_m}))$$

[5] in this paper, all but operators but $\oplus$ have arity 0 or 1

where $Mult(\phi, X)$ is the $FO_{\#}(\mathcal{M})$ formula stating that the multiplicity of each element of X satisfies ϕ (cf section 2). Therefore denoting by $L(TAC)$ the set of languages accepted by tree automata with arithmetic constraints, we get:

Proposition 6.4. $L(TAC) \subseteq CMTL$

[Lug98] defines a class of multitree automata which is stricly included in $CMTL$. The constraints of type 1 rules are only boolean combinations of equations where one side is a variable (e.g. $X_i = \Sigma_j \in \{1, \ldots, n\}X_j$) and the rules for terms of $\mathcal{S}$ can be replaced by type 2 rules where the constraint is a Presburger arithmetic formula. For instance, it is impossible to express normalization in this class, therefore it is disjoint from TAC. But due to the high expressive power of $FO_{\#}(\mathcal{M})$, both classes are included in $CMTL$. Moreover the algorithm to decide emptiness of $\mathcal{L}(\mathcal{A})$ used Dickson's lemma which prevented from stating that the complexity of the problem was elementary.

Feature Tree Automata. Feature tree automata have been introduced by Podelski and Niehren [NP93] to provide a notion of recognizable languages for feature trees. Feature trees can be seen as multisets constructed from a finite set of multiset constructors $\{,\}_A\{,\}_B \ldots$ and free unary symbols $f_1, f_2, \ldots$ (feature constructors)[6]. Recognizable sets are the multisets satisfying boolean combination of counting constraints of the form $\phi(\#(f_i))$. This kind of constraints is a very special case of $FO_{\#}(\mathcal{M})$ formula. In some sense, there are *local* constraints, since they don't relate the number of occurrences of some feature f and the number of occurrences of some other feature g. If we denote by $L(FTA)$ the set of multitree languages accepted by feature tree automata, we get:

Proposition 6.5. $L(FTA) \subseteq CMTL.$

Conclusion

One possible extension of this work is to look also at the associativity axiom. It is straightforward to add rules like hedge automata rules in this framework without losing properties, but a more interesting idea is to combine regularity constraints (like in hedge automata) and $FO_{\#}(\mathcal{M})$ formula. However we have found out that the resulting class is not closed under complement and doesn't enjoy determinization even when we allows only Presburger formulae [DL02]. Another possible question is to look for extensions allowing a bounded number of equality test along the acceptance process, but there is probably no satisfactory result to hope for in this direction since the relevant classes of automata are not closed under all the boolean operations, even when no associativity-commutativity axiom is used. Another direction of research is to use two-way tree automata (one can go up and down in the multitree). Some work has been

[6] In the original presentation of [NP93], the A's are constructors labelling nodes and the features f's label edges

done in this direction [GLV02], but undecidability quickly shows up and the counting/equality constraints that we use are probably too expressive to work well in that extension. Designing efficient implementations of our algorithms is also an issue: the main point is to balance the expressivity of constraints and a reasonable algorithmic efficiency using good data structures for multisets.

References

[Ber77] L. Berman. Precise bounds for Presburger arithmetic and the reals with addition: Preliminary report. In *Proc. of Symp. on Foundation Of Computer Science*, pages 95–99, 1977.

[BKN85] Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of matching problems. In *Proc. of 1st Int. Conf. on Rewriting techniques and Applications*, vol. 202 of *Lect. Notes in Comp. Sci.*, pages 417–429, 1985.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[BT92] B. Bogaert and S. Tison. Equality and disequality constraints on direct subterms in tree automata. In *Proc. of the 9th STACS*, vol. 577 of *Lect. Notes in Comp. Sci.*, pp. 161–172, 1992.

[CCC+94] A.C. Caron, H. Comon, J.L. Coquidé, M. Dauchet, and F. Jacquemard. Pumping, cleaning and symbolic constraints solving. In *Proc. 21st ICALP, Jerusalem (Israel)*, pages 436–449, 1994.

[CJ94] H. Comon and F. Jacquemard. Ground reducibility and automata with disequality constraints. In Springer-Verlag, editor, *Proc. of 11th STACS*, vol. 820 of *Lect. Notes in Comp. Sci.*, pages 151–162, 1994.

[Col02] Th. Colcombet. Rewriting in partial algebra of typed terms modulo aci. presented at the Infinity workshop, August 2002.

[DL02] S. DalZilio and D. Lugiez. Multitrees automata, Presburger's constraints and tree logics. Tech. Report 4631, INRIA, 2002.

[GLV02] J. Goubault-Larrecq and K.N. Verma. Alternating two-way AC-tree automata. Technical report, LSV, ENS Cachan, 2002.

[LM94] D. Lugiez and J.L. Moysset. Tree automata help one to solve equational formulae in AC-theories. *Journal of Symbolic Computation*, 18(4):297–318, 1994.

[Lug98] D. Lugiez. A good class of tree automata. In K. Larsen, S. Skyum, and G. Winskel, editors, *Proc. of 15th ICALP*, vol. 1443 of *Lect. Notes in Comp. Sci.*, pages 409–420. Springer-Verlag, 1998.

[Mur01] Makoto Murata. Extended path expression for XML. In ACM, editor, *Proc. of the 20th Symp. on Principles of Database Systems (PODS)*, Santa Barbara, USA, 2001. ACM.

[NP93] Joachim Niehren and Andreas Podelski. Feature automata and recognizable sets of feature trees. In *Proc. TAPSOFT'93*, vol. 668 of *Lect. Notes in Comp. Sci.*, pages 356–375, 1993.

[Ohs01] Hitoshi Ohsaki. Beyond the regularity: Equational tree automata for associative and commutative theories. In *CSL 2001*, vol. 2142 of *Lect. Notes in Comp. Sci.*. Springer-Verlag, 2001.

[PQ68] C. Pair and A. Quéré. Définition et étude des bilangages réguliers. *Information and Control*, 13(6):565–593, 1968.

Compositional Circular Assume-Guarantee Rules Cannot Be Sound and Complete

Patrick Maier

Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
maier@mpi-sb.mpg.de

Abstract. Circular assume-guarantee reasoning is used for the compositional verification of concurrent systems. Its soundness has been studied in depth, perhaps because circularity makes it anything but obvious. In this paper, we investigate completeness. We show that compositional circular assume-guarantee rules cannot be both sound and complete.

1 Introduction

The goal in compositional verification of concurrent systems is to prove that a complex system, a parallel composition of several subsystems, satisfies a complex property, a conjunction of several simpler properties. In principle, verification tools can attack such a goal directly, at least if the complex system is finite still. However, the state space of the system may be exponentially larger than that of any subsystem — a phenomenon called *state explosion* — which may cause verification to become intractable in practice. Compositional verification tries to use the modular structure of complex systems and properties to decompose intractable verification tasks into a bunch of smaller, hopefully tractable subtasks; ideally, each subtask only establishes properties of a single subsystem in isolation. Later, one deduces from all these subtasks via a suitable proof rule that the original complex system satisfies the desired complex property.

Systems, Properties. We model systems and properties uniformly as elements of $\mathbf{S} = \langle S, \wedge, 1, \leq \rangle$, a meet-semilattice with one, i. e., a partial order $\langle S, \leq \rangle$ with greatest element 1 in which the greatest lower bound $x \wedge y$ of any two elements x and y exists. In this model, the expression $x \leq y$ can have three different readings depending on whether x and y denote systems or properties, respectively. If both are systems then $x \leq y$ means that x refines y, if both are properties then it means that x entails y, and if x is a system and y a property then $x \leq y$ expresses that x satisfies y. Likewise, $x \wedge y$ denotes composition if x and y are systems, it denotes conjunction if x and y are properties, and if x is system and y a property then $x \wedge y$ — x constrained by y — is the coarsest refinement of x that satisfies y. Thus, all we require of systems (properties) is that refinement (entailment) is an order and that composition (conjunction) is an associative commutative and idempotent operation which respects the order. Section 4 will show that this

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 343–357, 2003.

abstract algebraic setting already suffices for proving incompleteness of circular assume-guarantee reasoning. In particular, no notion of computation is required, unlike in the proofs for soundness.

Example 1. Meet-semilattices are a natural model for systems and properties. For instance, in (linear-time) temporal verification, often one views systems and properties as languages over some non-empty (and possibly infinite) alphabet Σ. In this setting, refinement and entailment correspond to language inclusion, and composition and conjunction correspond to language intersection, so we have a meet-semilattice structure. How that meet-semilattice actually looks like, depends on the type of properties we want to verify.

By the characterization in [3], safety properties are expressible as prefix-closed $*$-languages. I. e., a safety property may be viewed as a subset L of Σ^* such that for all $w \in \Sigma^*$, if w belongs to L then all prefixes of w belong to L, too. So for verification of safety properties, the meet-semilattice is the set of prefix-closed $*$-languages (over Σ). Note that these are exactly the languages generated by (possibly infinite) labeled state transition graphs, which are a natural representation of systems. In general, when verifying arbitrary temporal properties, the meet-semilattice will be the set of ω-languages (over Σ), i. e., the power set of Σ^ω. Here, natural representations of systems and properties are some more elaborate variants of state transition graphs, for instance fair transition systems [14] or (possibly infinite state) ω-automata [19]. □

Proof Rules. In general, there are two kinds of proof rules for compositional verification, non-circular and circular ones. We show some examples to demonstrate the difference. Let $s_1, s_2 \in S$ be systems and $p_1, p_2 \in S$ properties and suppose we want to verify that the composition of s_1 and s_2 satisfies the conjunction of p_1 and p_2. (1) shows two non-circular rules for this purpose. Both rules decompose the goal into two subgoals, where the subgoals of the first rule state that system s_i satisfies property p_i, or in other words: s_i guarantees p_i. The second rule differs only in the second subgoal, which states that s_2 constrained by p_1 satisfies p_2, or in other words: if p_1 is assumed then s_2 guarantees p_2 — hence the wide-spread term *assume-guarantee rule*.

$$\frac{s_1 \leq p_1 \qquad s_2 \leq p_2}{s_1 \wedge s_2 \leq p_1 \wedge p_2} \qquad\qquad \frac{s_1 \leq p_1 \qquad p_1 \wedge s_2 \leq p_2}{s_1 \wedge s_2 \leq p_1 \wedge p_2} \tag{1}$$

Going one step further and also introducing an assumption in the first subgoal, we obtain the circular rule (2).

$$\frac{p_2 \wedge s_1 \leq p_1 \qquad p_1 \wedge s_2 \leq p_2}{s_1 \wedge s_2 \leq p_1 \wedge p_2} \tag{2}$$

Unlike the non-circular rules, (2) is unsound; for instance, if both systems are 1, the greatest element in $\mathbf{S}$, and both properties are equal and different from 1 then both premises hold but the conclusion does not. As soundness is indispensable, rule (2) must be restricted by a side condition which excludes cases as

the one above. Such circularity-breaking side conditions do exist; in fact, quite a number of restricted variants of (2) are proven sound (by induction usually) in the literature, see [1,4,11,17,20] to name just a few.

Completeness. As there are many variants of sound circular assume-guarantee rules, the question arises whether some are better than others. An important criterion for rating rules is the restrictiveness of the side condition; if the side condition is overly restrictive the rule is applicable to few cases only, hence it is considered worse than a variant with a less restrictive side condition (as long as that variant is sound still). In the best case, the rule is complete, i. e., the side condition is true whenever premises and conclusion are true. Thus, the side condition of a complete rule does not restrict the rule unnecessarily since it is true whenever the rule should be applicable. Note however, that the side condition is not redundant in complete rules; it may still be indispensable for proving soundness.

Compositionality. Recall that compositional verification seeks to reduce a large, intractable goal into many smaller subgoals. The rules in (1) and (2) support this approach as the premises of these rules are less complex than their conclusions. In particular, no premise involves the composition of the systems s_1 and s_2 any more, so verification of the subgoals is more likely to be tractable than direct verification of the goal. However, when using a circular rule which is restricted by a side condition, it does not suffice to verify the subgoals that arise from the premises; additionally, we need to prove that the side condition holds. It may be the case that this proof requires to consider both systems simultaneously — e. g., for establishing some mutual exclusion property — and thus involves some aspects of the composition, which is against the spirit of compositional verification. Therefore, a rule can be called *compositional* only if checking the side condition is possible without taking into account both systems simultaneously, i. e., only if the side condition is expressible as a boolean combination of sub-conditions, each of which involves at most one of the systems.

Plan. Section 2 formally presents proof rules which are restricted by a side condition and defines soundness and completeness. Section 3 specifies what we mean by circular assume-guarantee reasoning in the context of compositional verification and formalizes the precise requirements for rules to be compositional. Section 4 proves the main result that compositional circular assume-guarantee rules cannot be both sound and complete. Finally, Section 5 discusses related work and Section 6 concludes. Proofs which have been omitted here due to lack of space can be found in [13].

2 Inference Rules

Terms, Formulas. We fix a set of variables *Var*. Terms are built inductively from variables in *Var*, the nullary operator $\top$, called *top*, and the binary operator

$\sqcap$, called *meet*. We consider top neutral w. r. t. meet, which is seen as associative, commutative and idempotent. We call a term t *atomic* iff t is a variable or t is top. By $var(t)$, we denote the set of variables occuring in t, and we say that a term t' is a *subterm* of t iff $var(t') \subseteq var(t)$.

A formula φ is a pair $\langle t, t' \rangle$ of terms, written as $t \sqsubseteq t'$. We refer to t as the *left-*, to t' as the *right-hand side* of φ. We denote the set of variables occuring in φ by $var(\varphi)$, i. e., $var(\varphi) = var(t) \cup var(t')$, and for every set of formulas Φ, we define $var(\Phi) = \bigcup_{\varphi \in \Phi} var(\varphi)$.

Truth, Entailment. We fix $\mathbf{S} = \langle S, \wedge, 1, \leq \rangle$, a non-trivial meet-semilattice with one. By *Val*, we denote the set of *valuations*, i. e., the set of total functions from *Var* to S. We extend a valuation α to terms in the canonical way, i. e., $\alpha(\top) = 1$ and $\alpha(t_1 \sqcap t_2) = \alpha(t_1) \wedge \alpha(t_2)$. Note that we may view any term t as a total function from *Val* to S by defining the function application $t(\alpha)$ as $\alpha(t)$.

We say that a formula $t \sqsubseteq t'$ is *true* under a valuation α, denoted by $\alpha \models t \sqsubseteq t'$, iff $\alpha(t) \leq \alpha(t')$. We extend truth to sets of formulas, i. e., $\alpha \models \Phi$ iff $\alpha \models \varphi$ for all $\varphi \in \Phi$.

We say that Φ *entails* Ψ, denoted by $\Phi \models \Psi$, iff for all $\alpha \in Val$, $\alpha \models \Phi$ implies $\alpha \models \Psi$. We say that Φ is *equivalent* to Ψ, denoted by $\Phi \equiv \Psi$, iff $\Phi \models \Psi$ and $\Psi \models \Phi$. Note that in sets of formulas the operators top and meet are redundant on right-hand sides since for terms t, t'_1, t'_2, we have the equivalences $\{t \sqsubseteq \top\} \equiv \emptyset$ and $\{t \sqsubseteq t'_1 \sqcap t'_2\} \equiv \{t \sqsubseteq t'_1, t \sqsubseteq t'_2\}$.

Relations. Let X, Y and Z be sets and $n \in \mathbb{N}$. Given n functions $f_1, \ldots, f_n : X \to Y$ and an n-ary function $g : Y^n \to Z$, we define the *n-ary composition* of g and $f_1, \ldots, f_n$ as the function $g[f_1, \ldots, f_n] : X \to Z$ such that for all $x \in X$, $g[f_1, \ldots, f_n](x) = g(f_1(x), \ldots, f_n(x))$.

Let $n \in \mathbb{N}$, let $t_1, \ldots, t_n$ be n terms and let $C : S^n \to \{0, 1\}$, i. e., C is the characteristic function of an n-ary relation on S, the carrier of our fixed meet-semilattice $\mathbf{S}$. Viewing terms as functions from *Val* to S, the function $C[t_1, \ldots, t_n] : Val \to \{0, 1\}$ is well-defined — it is the characteristic function of some set of valuations — and we say that C is *associated* with the terms $t_1, \ldots, t_n$. Note that for every enumeration $x_1, \ldots, x_m$ of a superset of the variables occuring in the terms $t_1, \ldots, t_n$ there is a unique function $C' : S^m \to \{0, 1\}$ such that $C'[x_1, \ldots, x_m] = C[t_1, \ldots, t_n]$. Therefore, without loss of generality, we may assume that the associated terms are variables.

A relation Γ is an n-ary function $C : S^n \to \{0, 1\}$ associated with n variables $x_1, \ldots, x_n$, i. e., $\Gamma = C[x_1, \ldots, x_n]$. We denote the set of variables occuring in Γ by $var(\Gamma)$, i. e., $var(\Gamma) = \{x_1, \ldots, x_n\}$.

We define a notion of truth for relations, similar to the one for formulas. We say that a relation $C[x_1, \ldots, x_n]$ is *true* under a valuation α, denoted by $\alpha \models C[x_1, \ldots, x_n]$, iff $C[x_1, \ldots, x_n](\alpha) = 1$. Rewriting this with the definition of n-ary composition, we see that $\alpha \models C[x_1, \ldots, x_n]$ iff $C(\alpha(x_1), \ldots, \alpha(x_n)) = 1$. We say that a relation Γ is *true* iff $\alpha \models \Gamma$ for all $\alpha \in Val$. Note that every set of formulas Φ may be expressed by an equivalent relation Γ_Φ with $var(\Gamma_\Phi) = var(\Phi)$, where for all valuations α, $\alpha \models \Phi$ iff $\alpha \models \Gamma_\Phi$. However, relations are strictly more

expressive than formulas. For instance, inequality of two distinct variables x and y is not expressible by formulas, i. e., there is no set of formulas Φ such that for all $\alpha \in \mathit{Val}$, $\alpha \models \Phi$ iff $\alpha(x) \neq \alpha(y)$.

Inference Rules. An inference rule (or rule, for short) R is a triple $\langle \Phi, \psi, \Gamma \rangle$, where the *premises* Φ are a finite set of formulas, the *conclusion* ψ is a formula, and the *side condition* Γ is a relation. We say that R is *syntactic* iff the side condition Γ is true. We write a rule R as $R : \Phi/\psi$ if Γ or

$$R : \frac{\varphi_1 \ \cdots \ \varphi_m}{\psi} \text{ if } C[x_1, \ldots, x_n]$$

when $\Phi = \{\varphi_1, \ldots, \varphi_m\}$ and $\Gamma = C[x_1, \ldots, x_n]$. If R is syntactic then we may omit the side condition and write $R : \Phi/\psi$, simply. Without loss of generality we will assume that the right-hand sides of all premises are atomic and that $\mathit{var}(\psi) \cup \mathit{var}(\Gamma) \subseteq \mathit{var}(\Phi)$, i. e., every variable of the conclusion or the side condition occurs in the premises.

Soundness, Completeness. Let $R : \Phi/\psi$ if Γ be an inference rule. We say that R is *sound* iff for all valuations α, $\alpha \models \Phi$ and $\alpha \models \Gamma$ implies $\alpha \models \psi$. We say that R is *syntactically sound* iff $\Phi \models \psi$. Note that syntactical soundness implies soundness, and every sound syntactic rule is syntactically sound.

We say that R is *complete* iff for all valuations α, $\alpha \models \Phi$ and $\alpha \models \psi$ implies $\alpha \models \Gamma$. Note that every syntactic rule is complete, trivially. Also note that for syntactically sound rules completeness is not an issue, as every syntactically sound rule $R : \Phi/\psi$ if Γ can be transformed by omitting the side condition into the (sound and complete) syntactic rule $R' : \Phi/\psi$. Hence, there is no reason why a syntactically sound rule should be restricted by a side condition.

Example 2. Assume that $\mathbf{S} = \langle S, \wedge, 1, \leq \rangle$ is the four-element meet-semilattice which is not a chain. Let s_1, s_2, p_1 and p_2 be four distinct variables, where we think of the s_i as representing systems and of the p_j as representing properties. We define the rules R_1 and R_2, where

$$R_k : \frac{p_2 \sqcap s_1 \sqsubseteq p_1 \qquad p_1 \sqcap s_2 \sqsubseteq p_2}{s_1 \sqcap s_2 \sqsubseteq p_1 \sqcap p_2} \text{ if } C_k[s_1, s_2, p_1, p_2]$$

and for all $a, b, c, d \in S$, $C_1(a, b, c, d) = 1$ iff c and d are incomparable, and $C_2(a, b, c, d) = 1$ iff $a \wedge b \leq c$ and $a \wedge b \leq d$. Both rules are sound as both side conditions are restrictive enough to prevent unsound circular reasoning. R_2 is trivially complete as $C_2[s_1, s_2, p_1, p_2]$ is equivalent to the conclusion. However, R_1 is incomplete as, for instance, it is not applicable to the (trivial) case when both properties equal 1. Note that the relation $C_1[s_1, s_2, p_1, p_2]$ is not expressible by formulas, which demonstrates that the language of side conditions is more expressive than languages of premises and conclusions. □

$$R_3 : \frac{s_1 \sqsubseteq p_1 \qquad s_2 \sqsubseteq p_2}{s_1 \sqcap s_2 \sqsubseteq p_1 \sqcap p_2} \qquad R_6 : \frac{p_2 \sqcap s_1 \sqsubseteq p_1 \qquad p_1 \sqcap s_2 \sqsubseteq p_2}{s_1 \sqcap s_2 \sqsubseteq p_1 \sqcap p_2}$$

$$R_4 : \frac{s_1 \sqsubseteq p_1 \qquad p_1 \sqcap s_2 \sqsubseteq p_2}{s_1 \sqcap s_2 \sqsubseteq p_1 \sqcap p_2} \qquad R_7 : \frac{p_3 \sqcap s_1 \sqsubseteq p_1 \qquad p_1 \sqcap s_2 \sqsubseteq p_2}{s_1 \sqcap s_2 \sqsubseteq p_1 \sqcap p_2}$$

$$R_5 : \frac{p_3 \sqcap s_1 \sqsubseteq p_1 \qquad p_1 \sqcap s_2 \sqsubseteq p_2}{p_3 \sqcap s_1 \sqcap s_2 \sqsubseteq p_1 \sqcap p_2} \qquad R_8 : \frac{p_1 \sqcap s_1 \sqsubseteq \top \qquad p_2 \sqsubseteq s_2}{p_1 \sqcap s_1 \sqsubseteq s_2 \sqcap p_2}$$

Fig. 1. Sample assume-guarantee rules

3 Assume-Guarantee Rules

Assume-Guarantee Rules. We call an inference rule $R : \Phi/\psi$ if Γ an assume-guarantee rule (or A-G rule, for short) iff for all premises $\varphi \in \Phi$, the left-hand side of ψ and the right-hand side of φ do not share any variables. We call an A-G rule $R : \Phi/\psi$ if Γ *circular* iff $\Phi \not\models \psi$.

Example 3. As the definition of A-G rules does not involve the side condition, we may illustrate it using syntactic rules only, see figure 1. There, s_1, s_2, p_1, p_2 and p_3 are five distinct variables, where the system/property distinction is as in example 2.

The rules R_3, R_4 and R_5 are non-circular A-G rules. Note the second premise of R_4, which may be read as *assuming the property* p_1 *the system* s_2 *guarantees the property* p_2. Likewise, the conclusion of R_5 may be read as *assuming* p_3 *the composition of* s_1 *and* s_2 *guarantees both* p_1 *and* p_2. This should explain where the term *assume-guarantee rule* comes from.

The rules R_6, R_7 and R_8 (and also R_1 and R_2 from example 2) are circular A-G rules as they are not syntactically sound. The term *circular* is justified for R_6, whose premises express circular assume-guarantee dependencies between the properties p_1 and p_2. For R_7 and R_8, however, there is no circularity in the premises. In the case of R_7, unsoundness arises from the unresolved assumption p_3; compare to R_5 where that assumption is resolved. R_8 is unsound because it is nonsense, it serves to demonstrate that not every assume-guarantee rule has a meaningful reading. So, the term *circular* should not be taken literally, rather it is an abstraction capturing the most important property of circular assume-guarantee reasoning, namely its lack of syntactical soundness. □

The following propositions provide an alternative characterization of circularity resp. a sufficient criterion for the truth of the premises of an A-G rule.

Proposition 1. *An A-G rule* $R : \Phi/t_\psi \sqsubseteq t_\psi{}'$ if Γ *is circular if and only if* $\Phi \not\models t_\psi \sqsubseteq x$ *for some* $x \in var(t_\psi{}')$.

Proposition 2. *Let* $R : \Phi/t_\psi \sqsubseteq t_\psi{}'$ if Γ *be an A-G rule and let* α *be a valuation. If for all* $x, y \in var(\Phi) \setminus var(t_\psi)$,

– $\Phi \models t_\psi \sqsubseteq x$ *implies* $\alpha(x) = 1$, *and*
– $\Phi \not\models t_\psi \sqsubseteq x$ *and* $\Phi \not\models t_\psi \sqsubseteq y$ *implies* $\alpha(x) = \alpha(y)$,

then $\alpha \models \Phi$.

A-G Rules for Compositional Verification. As has already been hinted in example 3, for the purpose of verification we distinguish systems and properties, so we partition our variable set *Var* into *system variables* s_i and *property variables* p_j. Section 4 will show that already the composition of only two systems exhibits the incompleteness of compositional circular assume-guarantee reasoning, so actually we can restrict the variable set to $Var = \{s_1, s_2\} \uplus \{p_1, p_2, p_3, \dots\}$.

The goal of compositional verification is to establish that the composition of some systems (in our case, s_1 and s_2) guarantees some property (possibly assuming some other property). So for an A-G rule to be useful for compositional verification, $s_1 \sqcap s_2$ must be a subterm of the left-hand side of the conclusion, which we will implicitly assume henceforth. Thus, without loss of generality we may assume that an A-G rule R is presented in the form

$$R : \frac{\varphi_1 \quad \cdots \quad \varphi_m}{t_\psi \sqsubseteq t_\psi{}'} \text{ if } C[s_1, s_2, p_1, \dots, p_n]$$

where $\{s_1, s_2\} \subseteq var(t_\psi)$ and $var(\{\varphi_1, \dots, \varphi_m, t_\psi \sqsubseteq t_\psi{}'\}) = \{s_1, s_2, p_1, \dots, p_n\}$. The latter requirement can always be achieved by renaming some property variables and extending and reordering the associated variables in the side condition. By the definition of A-G rules, $var(t_\psi) \cap var(t') = \emptyset$ for every premise $t \sqsubseteq t'$, so $t' \in \{\top, p_1, \dots, p_n\}$ as we assume the right-hand sides of premises to be atomic.

Compositionality. Let $R : \Phi/\psi$ if $C[s_1, s_2, p_1, \dots, p_n]$ be an A-G rule. We will call R compositional if it avoids the system composition $s_1 \sqcap s_2$ in the premises as well as in the side condition. Formally, we say that R is *compositional in the premises* iff $s_1 \sqcap s_2$ is not a subterm of any left-hand side in Φ. We say that R is *compositional in the side condition* iff $C[s_1, s_2, p_1, \dots, p_n]$ is expressible as a boolean combination of relations whose associated variables either do not include s_1 or s_2. I. e., R is compositional in the side condition iff there are $r_1, r_2 \in \mathbb{N}$, a $(r_1 + r_2)$-ary boolean function $F : \{0,1\}^{r_1+r_2} \to \{0,1\}$ and $r_1 + r_2$ $(n+1)$-ary functions $C_1^1, \dots, C_1^{r_1}, C_2^1, \dots, C_2^{r_2} : S^{n+1} \to \{0,1\}$ such that

$$C[s_1, s_2, \tilde{p}] = F\big[C_1^1[s_1, \tilde{p}], \dots, C_1^{r_1}[s_1, \tilde{p}], C_2^1[s_2, \tilde{p}], \dots, C_2^{r_2}[s_2, \tilde{p}]\big] \tag{3}$$

where $\tilde{p}$ abbreviates the enumeration $p_1, \dots, p_n$. We say that R is *compositional* iff it is compositional in the premises and in the side condition.

One may think of the above functions C_i^k as abstracting the system s_i together with the properties $p_1, \dots, p_n$ to a boolean value. Actually, we can relax the above definition of compositionality in the side condition from boolean to arbitrary finitary abstractions C_i^k. I. e., R is compositional in the side condition iff there are a finite set D and $r_1, r_2 \in \mathbb{N}$ and $F : D^{r_1+r_2} \to \{0,1\}$ and $C_1^1, \dots, C_1^{r_1}, C_2^1, \dots, C_2^{r_2} : S^{n+1} \to D$ such that the equation (3) holds.

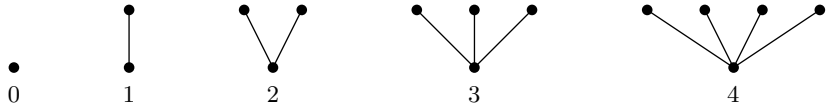


Fig. 2. Forks of width 0 to 4

Example 4. Recall the A-G rules R_1 to R_8 from the examples 2 and 3. All these rules are compositional in the premises, and the syntactic rules R_3 to R_8 are compositional in the side condition, trivially. The rule R_1 is also compositional in the side condition but R_2 is not. □

4 Incompleteness of Compositional Rules

Forks. We say that $Y \subseteq S$ is a *fork* iff there is $x \in Y$ such that for all $y, z \in Y$, $y \neq z$ implies $x = y \wedge z$; if Y is infinite then we say that Y is a fork of *infinite width*, otherwise the size of Y is $m \in \mathbb{N}$ and we say that Y is a fork of *width* $m-1$. Note that if $\mathbf{S}$ contains a fork of infinite width then it also contains forks of width m for every $m \in \mathbb{N}$.

Example 5. Some forks of finite width are depicted in figure 2. Note that if $\mathbf{S}$ is a chain then it contains only forks of width 1, and if $\mathbf{S}$ is the power set meet-semilattice of an arbitrary set X then it contains forks of infinite width iff X is infinite. In particular, the meet-semilattice of ω-languages over some alphabet Σ (see example 1) contains forks of infinite width iff Σ is not unary. The same holds for the meet-semilattice of prefix-closed $*$-languages over Σ. This is so because if Σ is unary then the prefix-closed $*$-languages form a chain. And if Σ contains the distinct letters a and b, then $Y = \{a^*\} \cup \{a^* \cup a^i b^* \mid i \in \mathbb{N}\}$ is a fork of infinite width. □

In order to prove our main theorem, we need two lemmas. Lemma 3 is purely combinatorial, it states the infeasibility of particular boolean equation systems. Lemma 4 forms the core of the main theorem. Provided that the semilattice $\mathbf{S}$ contains forks of sufficient width, it reduces the existence of a sound and complete circular A-G rule R which is compositional in the side condition to feasibility of a boolean equation system, which is known to be be infeasible by Lemma 3. This contradiction implies that R must be unsound or incomplete.

Lemma 3. *Let $m, n \in \mathbb{N}$ and $F : \{0,1\}^{m+n} \to \{0,1\}$. Then the system of equations E_F over the variables u_i^k $(0 \leq i \leq 2^{\min\{m,n\}}, 1 \leq k \leq m)$ and v_j^l $(0 \leq j \leq 2^{\min\{m,n\}}, 1 \leq l \leq n)$ has no solutions, where E_F is defined as*

$$E_F = \{F(u_i^1, \ldots, u_i^m, v_j^1, \ldots, v_j^n) = 1 \mid 0 \leq i, j \leq 2^{\min\{m,n\}}, i \neq j\}$$
$$\cup \{F(u_i^1, \ldots, u_i^m, v_i^1, \ldots, v_i^n) = 0 \mid 1 \leq i \leq 2^{\min\{m,n\}}\}.$$

Lemma 4. *Let* $R : \Phi/\psi$ if $C[s_1, s_2, p_1, \ldots, p_n]$ *be a circular A-G rule. Let* $r_1, r_2 \in \mathbb{N}$*, let* $F : \{0,1\}^{r_1+r_2} \to \{0,1\}$ *be a* $(r_1 + r_2)$*-ary boolean function and let* $C_1^1, \ldots, C_1^{r_1}, C_2^1, \ldots, C_2^{r_2} : S^{n+1} \to \{0,1\}$ *be* $(n+1)$*-ary functions such that*

$$C[s_1, s_2, \tilde{p}] = F\big[C_1^1[s_1, \tilde{p}], \ldots, C_1^{r_1}[s_1, \tilde{p}], C_2^1[s_2, \tilde{p}], \ldots, C_2^{r_2}[s_2, \tilde{p}]\big] \tag{4}$$

where $\tilde{p}$ *stands for* $p_1, \ldots, p_n$*. If* $\mathbf{S}$ *contains a fork of width* $2^{\min\{r_1,r_2\}}$ *then* R *is unsound or incomplete.*

Proof. Without loss of generality we may assume that there is $m \in \{0, \ldots, n\}$ such that for all $j \geq 1$, $\Phi \models t_\psi \sqsubseteq p_j$ iff $j \leq m$, where t_ψ is the left-hand side of ψ. In what follows, we sketch a proof by contradiction.

Assume that R is sound and complete and let $Y \subseteq S$ be a fork of width $2^{\min\{r_1,r_2\}}$, i. e., $Y = \{x_0, x_1, \ldots, x_{2^r}\}$ with $r = \min\{r_1, r_2\}$ such that for all $i, j \geq 0$ with $i \neq j$, $x_0 = x_i \wedge x_j$. By Lemma 3, we know that the system of equations

$$\begin{aligned} &\{F(u_i^1, \ldots, u_i^{r_1}, v_j^1, \ldots, v_j^{r_2}) = 1 \mid 0 \leq i, j \leq 2^{\min\{r_1,r_2\}}, i \neq j\} \\ \cup\, &\{F(u_i^1, \ldots, u_i^{r_1}, v_i^1, \ldots, v_i^{r_2}) = 0 \mid 1 \leq i \leq 2^{\min\{r_1,r_2\}}\} \end{aligned} \tag{5}$$

over the variables u_i^k and v_j^l $(0 \leq i, j \leq 2^{\min\{r_1,r_2\}}, 1 \leq k \leq r_1, 1 \leq l \leq r_2)$ has no solutions. However, we will show that there is a solution to (5), namely with $C_1^k(x_i, \widetilde{1}, \widetilde{x_0})$ resp. $C_2^l(x_j, \widetilde{1}, \widetilde{x_0})$ as the values of u_i^k resp. v_j^l, where $\widetilde{1}$ abbreviates the list $1, \ldots, 1$ of length m, and $\widetilde{x_0}$ abbreviates the list $x_0, \ldots, x_0$ of length $n-m$. For all $i, j \in \{0, \ldots, 2^{\min\{r_1,r_2\}}\}$, we define a valuation α_{ij} such that $\alpha_{ij}(s_1) = x_i$, $\alpha_{ij}(s_2) = x_j$, $\alpha_{ij}(p_1) = \cdots = \alpha_{ij}(p_m) = 1$ and $\alpha_{ij}(p_{m+1}) = \cdots = \alpha_{ij}(p_n) = x_0$.

First, let $i, j \in \{0, \ldots, 2^{\min\{r_1,r_2\}}\}$ with $i \neq j$. Then we can show $\alpha_{ij} \models \Phi$ (by Proposition 2) and $\alpha_{ij} \models \psi$. Hence by completeness, $\alpha_{ij} \models C[s_1, s_2, \tilde{p}]$, i. e., $C[s_1, s_2, \tilde{p}](\alpha_{ij}) = 1$, which by (4) expands to the equation

$$F\big(C_1^1(x_i, \widetilde{1}, \widetilde{x_0}), \ldots, C_1^{r_1}(x_i, \widetilde{1}, \widetilde{x_0}), C_2^1(x_j, \widetilde{1}, \widetilde{x_0}), \ldots, C_2^{r_2}(x_j, \widetilde{1}, \widetilde{x_0})\big) = 1.$$

Second, let $i \in \{1, \ldots, 2^{\min\{r_1,r_2\}}\}$. Then we can show $\alpha_{ii} \models \Phi$ (by Proposition 2) and $\alpha_{ii} \not\models \psi$ (using Proposition 1). Thus, soundness forces $\alpha_{ii} \not\models C[s_1, s_2, \tilde{p}]$, i. e., $C[s_1, s_2, \tilde{p}](\alpha_{ii}) = 0$, which by (4) expands to the equation

$$F\big(C_1^1(x_i, \widetilde{1}, \widetilde{x_0}), \ldots, C_1^{r_1}(x_i, \widetilde{1}, \widetilde{x_0}), C_2^1(x_i, \widetilde{1}, \widetilde{x_0}), \ldots, C_2^{r_2}(x_i, \widetilde{1}, \widetilde{x_0})\big) = 0.$$

Thus, the system of equations (5) indeed has a solution, which ends this proof by contradiction. □

We have shown that an A-G rule which is compositional in the side condition cannot be both sound and complete — provided that the semilattice $\mathbf{S}$ contains forks of sufficient width. The latter is the case trivially whenever $\mathbf{S}$ contains a fork of infinite width.

Theorem 5. *If* $\mathbf{S}$ *contains forks of infinite width then there exists no sound and complete compositional circular assume-guarantee rule.*

Proof. Follows from Lemma 4. □

5 Discussion of Related Work

Incompleteness of Other Rules. Our setting of inference rules in meet-semilattices is a very abstract one. Most circular A-G rules in the literature are presented in more concrete settings, i. e., they use more structure than just meet and order — and that extra structure is usually indispensable for proving soundness by a circularity-breaking induction. This raises the question to what extent our incompleteness result is relevant for such concrete rules.

We claim that most circular A-G rules can be transformed — preserving soundness and compositionality — into equivalent circular A-G rules in meet-semilattices which contain forks of infinite width. Incompleteness of the transformed rule then points out a defect of the original rule: There must be cases in which the original rule is not applicable although soundness is not in danger. Below, we will exemplify two such transformations.

Various circular A-G rules have been proposed for settings, where systems and properties are presented as some form of transition graphs enriched with input and output, e. g., Moore or Mealy machines [11,9] or Reactive Modules [4]. These rules establish certain refinement relations, e. g., trace containment or simulation, between compositions of transition graphs. Thereby, composition is a partial operation, which is defined only if the components satisfy some condition called compatibility. These compatibilities form an implicit side condition to the A-G rules, which is made explicit by the transformation. We demonstrate this in the following example by means of the transformation of a circular A-G rule for Moore machines.

Example 6. Let $\mathcal{X}$ be a finite set of variables, ranging over an arbitrary non-empty domain $\mathcal{D}$. A *Moore machine* M is a (possibly infinite) state transition graph with input variables $I_M \subseteq \mathcal{X}$ and output variables $O_M \subseteq \mathcal{X}$, where the nodes resp. edges of the graph are labeled by valuations of the output resp. input variables; for a formal definition see for instance [9,11]. Naturally, one associates a trace language $[\![M]\!] \subseteq \Sigma^*$ with M, where $\Sigma = \mathcal{D}^{\mathcal{X}}$ is the set of valuations of all variables. The parallel composition $M_1 \parallel M_2$ of two Moore machines M_1 and M_2 corresponds to language intersection, i. e., $[\![M_1 \parallel M_2]\!] = [\![M_1]\!] \cap [\![M_2]\!]$. Note that $M_1 \parallel M_2$ is defined only if M_1 and M_2 are compatible, i. e., O_{M_1} and O_{M_2} are disjoint.

For Moore machines with trace semantics, the following circular proof rule is known:

$$\frac{[\![P_2 \parallel S_1]\!] \subseteq [\![P_1]\!] \qquad [\![P_1 \parallel S_2]\!] \subseteq [\![P_2]\!]}{[\![S_1 \parallel S_2]\!] \subseteq [\![P_1 \parallel P_2]\!]} \tag{6}$$

where S_1, S_2, P_1, P_2 are Moore machines such that all parallel compositions in (6) are defined. We transform this rule into the A-G rule R_{Moore} for the meet-semilattice $\langle \mathcal{P}(\Sigma^*), \cap, \Sigma^*, \subseteq \rangle$, the power set of Σ^*:

$$R_{\text{Moore}} : \frac{p_2 \sqcap s_1 \sqsubseteq p_1 \qquad p_1 \sqcap s_2 \sqsubseteq p_2}{s_1 \sqcap s_2 \sqsubseteq p_1 \sqcap p_2} \text{ if } F\big[C[s_1], C[p_1], C[s_2], C[p_2]\big]$$

where $D = \mathcal{P}(\mathcal{X}) \uplus \{\bot\}$ is a finite set and $C : \mathcal{P}(\Sigma^*) \to D$ is a finitary abstraction mapping each $L \in \mathcal{P}(\Sigma^*)$ to the least set of output variables O_M such that M is a Moore machine with $[\![M]\!] = L$; if no such Moore machine exists then $C(L) = \bot$. The function $F : D^4 \to \{0, 1\}$ is defined by $F(O_{s_1}, O_{p_1}, O_{s_2}, O_{p_2}) = 1$ iff

$$O_{s_1} \neq \bot \text{ and } O_{p_1} \neq \bot \text{ and } O_{s_2} \neq \bot \text{ and } O_{p_2} \neq \bot$$
$$\text{and } O_{s_1} \cap O_{s_2} = O_{p_1} \cap O_{p_2} = O_{p_2} \cap O_{s_1} = O_{p_1} \cap O_{s_2} = \emptyset.$$

R_{Moore} is a compositional circular A-G rule according to Section 3. Circularity and compositionality in the premises are obvious. Compositionality in the side condition holds as obviously there exist functions $C_i^k : \mathcal{P}(\Sigma^*)^3 \to D$ such that

$$F\big[C[s_1], C[p_1], C[s_2], C[p_2]\big] = F\big[C_1^1[s_1, p_1, p_2], C_1^2[s_1, p_1, p_2], C_2^1[s_2, p_1, p_2], C_2^2[s_2, p_1, p_2]\big].$$

Moreover, soundness of R_{Moore} can be reduced to soundness of the original rule (6), and vice versa, so both rules are applicable in exactly the same cases. To see how soundness of R_{Moore} reduces to (6), consider the premises and side condition of R_{Moore} to be true under a valuation α. Then there are Moore machines S_i and P_j such that $[\![S_i]\!] = \alpha(s_i)$ and $[\![P_j]\!] = \alpha(p_j)$ and S_1 and S_2, P_1 and P_2, P_2 and S_1 as well as P_1 and S_2 are compatible, i. e., all parallel compositions in (6) are defined. Furthermore, we have $[\![P_2]\!] \cap [\![S_1]\!] \subseteq [\![P_1]\!]$ and $[\![P_1]\!] \cap [\![S_2]\!] \subseteq [\![P_2]\!]$, which by language intersection and soundness of (6) implies $[\![S_1 \parallel S_2]\!] \subseteq [\![P_1 \parallel P_2]\!]$, which in turn by language intersection implies that the conclusion of R_{Moore} is true under α. To show the converse reduction, let S_i and P_j be Moore machines such that all parallel compositions in (6) are defined, i. e., S_1 and S_2, P_1 and P_2, P_2 and S_1 as well as P_1 and S_2 are compatible. Obviously, soundness of (6) follows by language intersection and soundness of R_{Moore}.

As the proof rule (6) has been proven sound in [11], R_{Moore} is a sound and compositional circular A-G rule. Thus by Theorem 5, R_{Moore} is incomplete because the meet-semilattice $\langle \mathcal{P}(\Sigma^*), \cap, \Sigma^*, \subseteq \rangle$ contains forks of infinite width. Hence there are cases in which circular reasoning is admissible yet the rule (6) is not applicable, due to partiality of parallel composition. □

Other kinds of circular A-G rules focus on temporal logics to present properties (and sometimes systems also), see for instance [1,2,10]. In order to break the circularity, such rules usually employ so-called assume-guarantee specifications, i. e., formulas of the form $\varphi \triangleright \psi$ where $\triangleright$ is a special temporal operator ensuring that during any computation the guarantee ψ holds at least one step longer than the assumption φ. In our meet-semilattice setting, we cannot express A-G specifications in the premises of inference rules. However, we can move A-G specifications to the side condition, where their truth is expressible as a relation. In the following example, we demonstrate this transformation on a simple circular rule for A-G specifications.

Example 7. Let AP be a non-empty set of atomic propositions. We say that $\Sigma = \mathcal{P}(AP)$ is the set of states, and Σ^ω is the set of computations. A system

is a set of computations, and the parallel composition of two systems S_1 and S_2 is their intersection $S_1 \cap S_2$. Likewise, a property is a set of computations, and we say that a property P entails another property Q iff $P \subseteq Q$. We may represent certain properties by formulas in linear-time temporal logic (LTL), which are constructed from atomic propositions by means of boolean operators and the standard temporal operators X (next-time), U (until), F (eventually) and G (always); for a formal definition of syntax and semantics of LTL see for instance [5]. Henceforth, we will identify a formula φ with the property it represents.

Given two formulas φ and ψ, we define the *assume-guarantee specification* $\varphi \triangleright \psi$ as an abbreviation of the formula $\neg(\varphi \mathsf{U} \neg\psi)$, cf. [18]. The temporal operator $\triangleright$ satisfies the following (in)equalities:

$$\varphi \triangleright \psi = \psi \wedge \big(\varphi \Rightarrow \mathsf{X}(\varphi \triangleright \psi)\big) \tag{7}$$

$$\mathsf{G}\varphi \wedge (\varphi \triangleright \psi) \subseteq \mathsf{G}\psi \tag{8}$$

From the fix-point equation (7), we can read off that $\varphi \triangleright \psi$ is the weakest property where ψ holds strictly longer than φ along every computation.

For A-G specifications, the following circular proof rule is known:

$$\frac{S_1 \subseteq \varphi_2 \triangleright \varphi_1 \qquad S_2 \subseteq \varphi_1 \triangleright \varphi_2}{S_1 \cap S_2 \subseteq \mathsf{G}(\varphi_1 \wedge \varphi_2)} \tag{9}$$

where S_1, S_2 are systems and φ_1, φ_2 are LTL formulas. We transform this rule into the A-G rule $R_\triangleright$ for the meet-semilattice of systems and properties $\langle \mathcal{P}(\Sigma^\omega), \cap, \Sigma^\omega, \subseteq \rangle$:

$$R_\triangleright : \frac{p_4 \sqcap s_1 \sqsubseteq p_3 \qquad p_3 \sqcap s_2 \sqsubseteq p_4}{s_1 \sqcap s_2 \sqsubseteq p_3 \sqcap p_4} \text{ if } C_1^1[s_1, p_1, \ldots, p_4] * C_2^1[s_2, p_1, \ldots, p_4]$$

where $* : \{0,1\}^2 \to \{0,1\}$ denotes multiplication (i. e., conjunction in logical terms) and the functions $C_i^k : \mathcal{P}(\Sigma^\omega)^5 \to \{0,1\}$ are defined by

$$C_1^1(S_1, P_1, P_2, P_3, P_4) = 1 \text{ iff } P_3 = \mathsf{G}P_1 \text{ and } P_4 = \mathsf{G}P_2 \text{ and } S_1 \subseteq P_2 \triangleright P_1,$$
$$C_2^1(S_2, P_1, P_2, P_3, P_4) = 1 \text{ iff } P_3 = \mathsf{G}P_1 \text{ and } P_4 = \mathsf{G}P_2 \text{ and } S_2 \subseteq P_1 \triangleright P_2.$$

Note that the equality $P_3 = \mathsf{G}P_1$ is supposed to hold iff there exists an LTL formula φ_1 such that φ_1 and $\mathsf{G}\varphi_1$ represent the properties P_1 and P_3, respectively; $P_4 = \mathsf{G}P_2$ is to be interpreted similarly.

Obviously, $R_\triangleright$ is a compositional circular A-G rule[1] according to Section 3. Moreover, soundness of $R_\triangleright$ can be reduced to soundness of the original rule (9), and vice versa, so both rules are applicable in exactly the same cases. To see how soundness of $R_\triangleright$ reduces to (9), consider the premises and side condition of $R_\triangleright$ to be true under a valuation α. Then there are LTL formulas φ_j such that $\mathsf{G}\varphi_1 = \alpha(p_3)$ and $\mathsf{G}\varphi_2 = \alpha(p_4)$ and $\alpha(s_1) \subseteq \varphi_2 \triangleright \varphi_1$ and $\alpha(s_2) \subseteq \varphi_1 \triangleright \varphi_2$. Using soundness of (9), we infer $\alpha(s_1) \cap \alpha(s_2) \subseteq \mathsf{G}(\varphi_1 \wedge \varphi_2)$, which implies that the

[1] The trivial premises $p_1 \sqsubseteq \top$ and $p_2 \sqsubseteq \top$ have been omitted from the definition of $R_\triangleright$ for the sake of readability.

conclusion of $R_{\triangleright}$ is true under α. To show the converse reduction, let S_i and φ_j be systems and formulas, respectively, such that the premises of rule (9) hold. By (8), these premises imply $\mathsf{G}\varphi_2 \cap S_1 \subseteq \mathsf{G}\varphi_1$ and $\mathsf{G}\varphi_1 \cap S_2 \subseteq \mathsf{G}\varphi_2$, respectively. Using soundness of $R_{\triangleright}$, we infer $S_1 \cap S_2 \subseteq \mathsf{G}\varphi_1 \cap \mathsf{G}\varphi_2$, which is equivalent to the conclusion of (9).

As the proof rule (9) is sound, cf. [16,18], $R_{\triangleright}$ is a sound and compositional circular A-G rule. Thus by Theorem 5, $R_{\triangleright}$ is incomplete because the meet-semilattice $\langle \mathcal{P}(\Sigma^\omega), \cap, \Sigma^\omega, \subseteq \rangle$ contains forks of infinite width. As a consequence, the rule (9) does not capture all sound circular reasoning patterns, i. e., there are cases in which circular reasoning is admissible yet (9) is not applicable. □

In short, this paper shows that compositionality implies incompleteness. Yet, we did not encounter any complete rule except for the rather trivial rule R_2 from example 2. This raises the question whether non-trivial sound and complete circular A-G rules do exist at all. They do — in [12], we present a very general sound and complete circular A-G rule for certain classes of lattices. Of course, that rule must be non-compositional; in fact, it is non-compositional both in the premises and in the side condition. Still, that general rule can be instantiated to many known circular A-G rules, no matter whether they are compositional or not.

Other Notions of Completeness. When some complex system should be verified against a conjunction of properties, one usually applies backward reasoning, i. e., one matches the verification goal against the conclusion of a proof rule and from the premises and the side condition one infers the subgoals that need to be established. In [18], the authors investigate a notion of completeness that characterizes rules which always enable backward reasoning, so we will term this notion backward completeness. Adopted to our setting, a rule $R : \Phi/\psi$ if Γ is called *backward complete* iff for all valuations α, $\alpha \models \psi$ implies $\alpha' \models \Phi$ and $\alpha' \models \Gamma$ for some valuation α' which agrees with α on the variables of ψ. Thus, truth of the conclusion implies that the premises and the side condition can be made true through choosing (i. e., guessing) appropriate values for the *auxiliary variables*, i. e., for those variables in the premises that do not occur in the conclusion. Note that backward completeness does not distinguish premises and side condition, whereas this distinction is essential for our notion of completeness.

Our notion of completeness relates more to forward reasoning, i. e., from prior knowledge which subsystems guarantee which properties assuming which other properties, we want to infer that the complex system guarantees a conjunction of properties. A complete rule (in the sense of this paper) will enable this inference whenever the conclusion is consistent with our knowledge. Still, our incompleteness result bears some significance for backward complete rules. For a rule $R : \Phi/\psi$ if Γ without auxiliary variables, i. e., $var(\psi) = var(\Phi)$, backward completeness implies completeness. Thus, as a consequence of Theorem 5, every sound and backward complete compositional circular A-G rule necessarily needs to employ auxiliary variables. In other words, backward reasoning with compositional circular rules is likely to require guessing auxiliary assertions about the

system. I. e., one trades the lower complexity of the (decomposed) system for a higher complexity of the proof search.

6 Conclusion

We have shown that sound and compositional circular assume-guarantee rules, presented as inference rules restricted by an arbitrary side condition, cannot be complete. I. e., the side condition of a compositional rule, no matter how elaborate it is, cannot capture all cases where circular reasoning is admissible. Consequently, two important criteria for rating the quality of inference rules work against each other in the realm of circular reasoning. Upon designing assume-guarantee rules, this raises the question whether we should settle for compositionality or rather for completeness. The answer depends on the intended use of the rule.

Over the years, the practicality of circular assume-guarantee reasoning as a technique for compositional verification has been documented in a number of case studies, see [7,8,15] to name a few. In most cases, these assume-guarantee rules were tailored for model checking, and as model checkers particularly suffer from the infamous *state explosion* problem, the designers of the rules focussed on (automatic) system decomposition rather than on completeness. Consequently, these rules avoid to generate subgoals that involve a composition of subsystems. Here, compositional rules whose side conditions can be checked efficiently (but are not too restrictive) seem to be very appropriate. There are tools that successfully employ such incomplete compositional rules, e. g., in the verification of thread-parallel software [6]. To some extent, the loss of completeness can be mitigated against by human interaction, e. g., in the form of auxiliary annotations (to the code of the system), which provide more information about the system so the tools may find better decompositions.

To the best of our knowledge, there is no data available on the practical use of complete circular assume-guarantee rules in verification. However, in the case of manual (or almost manual) verification, we see no reason for severely restricting the power of circular reasoning, so one might prefer a complete rule over a compositional one. Of course, then one must tackle system decomposition in the subgoals by other means, e. g., by abstraction. Still, assume-guarantee reasoning may be superior to direct verification, as the additional assumptions in the subgoals may enable better abstractions.

Acknowledgement. The author thanks Viorica Sofronie-Stokkermans and Andreas Podelski for helpful discussions and comments and Carsten Sinz for support with theorem provers.

References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

2. M. Abadi and S. Merz. An abstract account of composition. In *MFCS*, LNCS 969, pages 499–508. Springer, 1995.
3. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
4. R. Alur and T. A. Henzinger. Reactive modules. In *LICS*, pages 207–218. IEEE Computer Society, 1996.
5. E. A. Emerson. Modal and temporal logics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 135–191. Elsevier, 1990.
6. C. Flangan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *ESOP*, LNCS 2305, pages 262–277. Springer, 2002.
7. T. A. Henzinger, X. Liu, S. Qadeer, and S. K. Rajamani. Formal specification and verification of a dataflow processor array. In *ICCAD*, pages 494–499. IEEE Computer Society, 1999.
8. T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV*, LNCS 1427, pages 440–451. Springer, 1998.
9. T. A. Henzinger, S. Qadeer, S. K. Rajamani, and S. Tasiran. An assume-guarantee rule for checking simulation. *ACM Transactions on Programming Languages and Systems*, 24(1):51–64, 2002.
10. B. Jonsson and Y.-K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167(1–2):47–72, 1996.
11. P. Maier. A set-theoretic framework for assume-guarantee reasoning. In *ICALP*, LNCS 2076, pages 821–834. Springer, 2001.
12. P. Maier. *A Lattice-Theoretic Framework For Circular Assume-Guarantee Reasoning*. PhD thesis, Universität des Saarlandes, 2002. Submitted.
13. P. Maier. Compositional circular assume-guarantee rules cannot be sound and complete. Technical Report MPI-I-2003-2-001, Max-Planck-Institut für Informatik, 2003.
14. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.
15. K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV*, LNCS 1427, pages 110–121. Springer, 1998.
16. K. L. McMillan. Circular compositional reasoning about liveness. In *CHARME*, LNCS 1703, pages 342–345. Springer, 1999.
17. J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.
18. K. S. Namjoshi and R. J. Trefler. On the completeness of compositional reasoning. In *CAV*, LNCS 1855, pages 139–153. Springer, 2000.
19. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 135–191. Elsevier, 1990.
20. M. Viswanathan and R. Viswanathan. Foundations for circular compositional reasoning. In *ICALP*, LNCS 2076, pages 835–847. Springer, 2001.

A Monadic Multi-stage Metalanguage

Eugenio Moggi and Sonia Fagorzi*

DISI, Univ. of Genova, v. Dodecaneso 35, 16146 Genova, Italy

Abstract. We describe a metalanguage MMML, which makes explicit the order of evaluation (in the spirit of monadic metalanguages) and the staging of computations (as in languages for multi-level binding-time analysis). The main contribution of the paper is an operational semantics which is sufficiently detailed for analyzing subtle aspects of multi-stage programming, but also intuitive enough to serve as a reference semantics. For instance, the separation of computational types from code types, makes clear the distinction between a computation for generating code and the generated code, and provides a basis for *multi-lingual* extensions, where a variety of programming languages (aka monads) coexist. The operational semantics consists of two parts: local (semantics preserving) simplification rules, and computation steps executed in a deterministic order (because they may have side-effects). We focus on the computational aspects, thus we adopt a simple type system, that can detect usual type errors, but not the *unresolved link* errors. Because of its explicit annotations, MMML is suitable as an intermediate language.

1 Introduction

Staging a computation into multiple steps is a well-known optimization technique used in algorithms, which exploits information available in early stages for generating code that will be executed in later stages. Multi-stage programming languages, like MetaML (see [9, 15, 16, 3]), provide constructs for expressing staging in a natural and concise manner, and must allow arbitrary interleaving of *code generation* and *computation*. Multi-stage programming is particularly convenient for defining *generative components*, which take as input a specification of user requirements and generate on the fly a customized component, or *mobile applications*, which need to adapt after each move, e.g. by assembling components downloaded remotely to generate code tailored to the local environment.

So far most of the theoretical research on multi-stage programming languages has focused on type systems (for the most recent proposals see [3, 10, 11]). The resulting operational semantics are often instrumental to a particular type system (thus difficult to relate and compare), and often ignore the subtle interactions between code generation and computational effects. In this paper, we provide a deeper understanding of the computational aspects of multi-stage programming, in the framework of a metalanguage with computational types $M\tau$

* Supported by MIUR project NAPOLI and EU project DART IST-2001-33477.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 358–374, 2003.

and code types $\langle\tau\rangle$: computational types classify terms describing computations, while code types classify terms representing other terms. We believe that in this framework one can have a fresh look at typing issues, and above all a *generic* approach for adding staging to a programming language (described in a monadic style), including a multi-lingual metalanguage.

An important principle of Haskell [12] is that pure functional evaluation (and all the optimization techniques that come with it) should not be corrupted by the addition of computational effects. In Haskell this separation has been achieved through the use of monads (like monadic IO and monadic state). When describing MMML we adopt this principle not only at the level of types, but also at the level of the operational semantics. In fact, we distinguish between *simplification* (described by local rewrite rules) and *computation* (that may cause side-effects).

Summary. Section 2 describes a general pattern for specifying the operational semantics of monadic metalanguages, which distinguishes simplification from computation. Section 3 exemplifies the general pattern by considering a monadic metalanguage MML for imperative computations. Section 4 introduces an extension MMML with staging, and explains how definitions and results for MML have to be modified and extended. Section 5 gives simple examples of MMML programs, which illustrate the most subtle points of the operational semantics. Section 6 discusses related work and issues specific to MMML.

Notation. In the paper we use the following notations and conventions.

- m, n range over the set N of natural numbers. Furthermore, $m \in \mathsf{N}$ is identified with the set $\{i \in \mathsf{N} | i < m\}$ of its predecessors.
- $\overline{e}$ ranges over the set E^* of finite sequences $(e_i | i \in m)$ of elements of E, and $|\overline{e}|$ denotes its length (i.e. , m). $\overline{e}_1, \overline{e}_2$ denotes the concatenation of $\overline{e}_1$ and $\overline{e}_2$.
- Term equivalence, written $\equiv$, is α-conversion. $\mathrm{FV}(e)$ is the set of variables free in e. If E is a set of terms, then E_0 is the set of $e \in \mathsf{E}$ s.t. $\mathrm{FV}(e) = \emptyset$. $e[x_i := e_i | i \in m]$ (and $e[\overline{x} := \overline{e}]$) denotes parallel substitution (modulo $\equiv$).
- $f : A \stackrel{fin}{\rightarrow} B$ means that f is a partial function from A to B with a finite domain, written $\mathsf{dom}(f)$. We write $\{a_i : b_i | i \in m\}$ for the partial function mapping a_i to b_i (where the a_i must be different, i.e. $a_i = a_j$ implies $i = j$). We use the following operations on partial functions: $\emptyset$ is the everywhere undefined partial function; f_1, f_2 denotes the union of two partial functions with disjoint domains; $f\{a : b\}$ denotes the extension of f to $a \notin \mathsf{dom}(f)$; $f\{a = b\}$ denotes the update of f in $a \in \mathsf{dom}(f)$.
- Given a BNF $e ::= P_1 \mid \ldots \mid P_m$, we write $e += P_{m+1} \mid \ldots \mid P_{m+n}$ as a shorthand for the extended BNF $e ::= P_1 \mid \ldots \mid P_{m+n}$.
- We write $\stackrel{*}{\longrightarrow}$ for the reflexive and transitive closure of a a relation $\longrightarrow$.

2 Monadic Metalanguages, Simplification and Computation

We outline a general pattern for specifying the operational semantics of monadic metalanguages, which distinguishes between *transparent simplification* and *programmable computation*. This is possible because in a monadic metalanguage there is a clear distinction between term-constructors for building terms of computational types, and the other term-constructors that are *computationally irrelevant*. For computationally relevant term-constructors we give an operational semantics that ensures the correct sequencing of computational effects, e.g. by adopting some well-established technique for specifying the operational semantics of programming languages (see [19]), while for computationally irrelevant term-constructors it suffices to give local simplification rules, that can be applied non-deterministically (because they are semantic preserving).

Remark 1. In [18] Wadler adopts a similar style, that distinguishes pure from monadic reduction. However, his pure reduction is a deterministic strategy, while simplification is non-deterministic. In this respect, our approach is related to the Cham [2]: simplification corresponds to heating and computation to reaction.

Combinatory Reduction Systems. We work in the setting of Combinatory Reduction Systems (CRS) [8], which extends Term Rewriting Systems (TRS) with binders. In Section 4 the uniformity of CRS descriptions is exploited for defining the extension with staging *generically* and concisely. In a CRS the syntax of terms is specified by a set C of term-constructors with given arity $\#\colon \mathsf{C} \to \mathsf{N}^*$

$$\boxed{e \in \mathsf{E} ::= x \mid c([\overline{x}_i]e_i | i \in m) \quad \text{with } \#c = (n_i | i \in m) \text{ and } \forall i \in m. |\overline{x}_i| = n_i}$$

Variables x belong to an infinite set X. More complex terms are built by applying a term-constructor c to a sequence of abstractions $[\overline{x}_i]e_i$ binding the free occurrences of the $\overline{x}_i$ in e_i, thus the set of free variables in $c([\overline{x}_i]e_i | i \in m)$ is

$$\mathrm{FV}(c([\overline{x}_i]e_i | i \in m)) \stackrel{\Delta}{=} \cup\{\mathrm{FV}([\overline{x}_i]e_i) | i \in m\} \quad \text{where } \mathrm{FV}([\overline{x}]e) = \mathrm{FV}(e) - \{\overline{x}\}$$

In CRS rewrite rules $e \longrightarrow e'$ can be specified as in TRS, for instance the β-rule is $@(\lambda([x]e'), e) \longrightarrow e'[x := e]$, where e and e' are arbitrary terms. It is possible to give a more schematic syntax for rewrite rules, but it requires metavariables ranging over abstractions.

Given a set T of types τ, a type system deriving judgments of the form $\Gamma \vdash e\colon \tau$, where $\Gamma\colon \mathsf{X} \stackrel{fin}{\to} \mathsf{T}$ is a type assignment, is specified by assigning to each term-constructor c of arity $\#c = (n_i | i \in m)$ a set of type schema $(\overline{\tau}_i \Rightarrow \tau_i | i \in m) \Rightarrow \tau$ consistent with $\#c$, i.e. , $|\overline{\tau}_i| = n_i$ for $i \in m$. More precisely, the typing rules are

$$x \ \frac{}{\Gamma \vdash x\colon \tau}\ \Gamma(x) = \tau \qquad \mathrm{c}\ \frac{\{\Gamma \vdash [\overline{x}_i]e_i\colon \overline{\tau}_i \Rightarrow \tau_i \mid i \in m\}}{\Gamma \vdash c([\overline{x}_i]e_i | i \in m)\colon \tau}\ c\colon (\overline{\tau}_i \Rightarrow \tau_i | i \in m) \Rightarrow \tau$$

where $\Gamma \vdash [\overline{x}]e\colon \overline{\tau} \Rightarrow \tau$ stands for $\Gamma, \{x_i\colon \tau_i | i \in m\} \vdash e\colon \tau$ with $\overline{x} = (x_i | i \in m)$ and $\overline{\tau} = (\tau_i | i \in m)$. Note that $\overline{\tau} \Rightarrow \tau$ is used in type schema, but it is not a type $\tau \in \mathsf{T}$.

Monadic Metalanguages. To specify a monadic metalanguage we define:

- Types $\tau \in \mathsf{T}$, including computational types $M\tau$.
- Terms $e \in \mathsf{E}$, including return $ret(e)$ and monadic do $do(e_1, [x]e_2)$, which corresponds to Haskell do-notation $x \Leftarrow e_1; e_2$.
- A type system, which amounts to give for each term-constructor a set of type schema, in particular for ret and do the type schema are $ret\colon \tau \Rightarrow M\tau$ and $do\colon M\tau_1, (\tau_1 \Rightarrow M\tau_2) \Rightarrow M\tau_2$
- A simplification relation $e \longrightarrow e'$ on terms, namely the *compatible closure* of a set of rewrite rules. By definition of $\longrightarrow$, the induced equivalence is always a congruence. In addition, we require that $\longrightarrow$ satisfies the Church Rosser (CR) and Subject Reduction (SR) properties.
- A *computation* relation $\longmapsto$ on *configurations*. A configuration $Id \in \mathsf{Conf}$ describes the state of a *closed system*, while the relation $\longmapsto$ describes how a closed system may evolve. Usually there is an obvious way to extend $\longrightarrow$ to configurations (preserving the CR property). To formulate a type safety result (along the lines of [19]), we must define well-formed configurations $\vdash Id$, show that both $\longrightarrow$ and $\longmapsto$ preserve well-formedness (for $\longrightarrow$ it should be an easy consequence of SR), and finally establish a progress property for $\Longrightarrow \stackrel{\Delta}{=} \longrightarrow \cup \longmapsto$.

Simplification should be *orthogonal* to computation, i.e. , if $Id_1 \stackrel{*}{\longrightarrow} Id_1'$ and Id_1 can move $Id_1 \longmapsto Id_2$, then Id_1' has a move $Id_1' \longmapsto Id_2'$ s.t. $Id_2 \stackrel{*}{\longrightarrow} Id_2'$.

3 MML: A Monadic Metalanguage for Imperative Computations

We introduce a monadic metalanguage MML for imperative computations, which exemplifies the pattern outlined in Section 2 in a *familiar case*, namely a subset of Haskell with the IO-monad. Moreover, MML provides a starting point for the addition of staging.

- Types $\boxed{\tau \in \mathsf{T} ::= \mathsf{nat} \mid M\tau \mid \tau_1 \to \tau_2 \mid \mathsf{ref}\ \tau}$. The type nat of natural numbers avoids a degenerate BNF (we will ignore it most of the time).
- Term-constructors $\boxed{c \in \mathsf{C} ::= ret \mid do \mid \lambda \mid @ \mid new \mid get \mid set \mid l}$. Locations l belong to an infinite set L (they are not allowed in user-written programs, but are instrumental to the operational semantics). The type schema for term-constructors (from which one can infer also their arity) are
  - $ret\colon \tau \Rightarrow M\tau$ and $do\colon M\tau_1, (\tau_1 \Rightarrow M\tau_2) \Rightarrow M\tau_2$
  - $@\colon (\tau_1 \to \tau_2), \tau_1 \Rightarrow \tau_2$ and $\lambda\colon (\tau_1 \Rightarrow \tau_2) \Rightarrow (\tau_1 \to \tau_2)$
  - $new\colon \tau \Rightarrow M(\mathsf{ref}\ \tau)$, $get\colon \mathsf{ref}\ \tau \Rightarrow M\tau$ and $set\colon \mathsf{ref}\ \tau, \tau \Rightarrow M(\mathsf{ref}\ \tau)$

  a signature $\Sigma\colon \mathsf{L} \stackrel{fin}{\to} \mathsf{T}$ gives the type to locations, i.e. , $l\colon \mathsf{ref}\ \tau$ when $\Sigma(l) = \tau$.

– The BNF for terms $e \in \mathsf{E}$ generated by the term-constructors above is
$\boxed{e ::= x \mid ret(e) \mid do(e_1, [x]e_2) \mid \lambda([x]e) \mid @(e_1, e_2) \mid new(e) \mid get(e) \mid set(e_1, e_2) \mid l}$
$\lambda([x]e)$ and $@(e_1, e_2)$ are λ-abstraction $\lambda x.e$ and application $e_1 e_2$; *new*, *get* and *set* are the ML-like operations $\mathsf{ref}\ e$, $!e$ and $e_1 := e_2$ on references.

The type system is parametric in Σ, and the rules for deriving judgments of the form $\Gamma \vdash_\Sigma e : \tau$ are

$$x\ \frac{}{\Gamma \vdash_\Sigma x : \tau}\ \Gamma(x) = \tau \qquad l\ \frac{}{\Gamma \vdash_\Sigma l : \mathsf{ref}\ \tau}\ \Sigma(l) = \tau$$

$$\mathrm{c}\ \frac{\{\Gamma \vdash_\Sigma [\overline{x}_i]e_i : \overline{\tau}_i \Rightarrow \tau_i \mid i \in m\}}{\Gamma \vdash_\Sigma c([\overline{x}_i]e_i \mid i \in m) : \tau}\ c : (\overline{\tau}_i \Rightarrow \tau_i \mid i \in m) \Rightarrow \tau$$

Simplification $\longrightarrow$ is the compatible closure of $@(\lambda([x]e_2), e_1) \longrightarrow e_2[x := e_1]$, i.e. , β-reduction. We write $=$ for β-equivalence, i.e. , the reflexive, symmetric and transitive closure of $\longrightarrow$. We recall the properties of simplification (β-reduction) relevant for our purposes.

Proposition 1 (Congr). *The equivalence $=$ induced by $\longrightarrow$ is a congruence.*

Proposition 2 (CR). *The simplification relation $\longrightarrow$ is confluent.*

Proposition 3 (SR). *If $\Gamma \vdash_\Sigma e : \tau$ and $e \longrightarrow e'$, then $\Gamma \vdash_\Sigma e' : \tau$.*

Remark 2. Several extensions can be handled at the level of simplification.

– The extension with a datatype, like nat or $\tau_1 \times \tau_2$, amounts to add term-constructors for introduction and elimination ($zero : \mathsf{nat}$, $succ : \mathsf{nat} \Rightarrow \mathsf{nat}$ and $case : \mathsf{nat}, \tau, (\mathsf{nat} \Rightarrow \tau) \Rightarrow \tau$) and simplification rules describing how they interact ($case(zero, e_0, [x]e_1) \longrightarrow e_0$ and $case(succ(e), e_0, [x]e_1) \longrightarrow e_1[x := e]$).
– Recursive definitions can be handled by a term-constructor $\mathsf{fix} : (\tau \Rightarrow \tau) \Rightarrow \tau$ with simplification rule $\mathsf{fix}([x]e) \longrightarrow e[x := \mathsf{fix}([x]e)]$. However, if one wants simplification of well-typed terms to terminate, then the type schema for fix should be $(M\tau \Rightarrow M\tau) \Rightarrow M\tau$ and $\mathsf{fix}([x]e)$ becomes a *computation redex*.
– A test for equality of references $\mathsf{ifeq} : \mathsf{ref}\ \tau, \mathsf{ref}\ \tau, \tau', \tau' \Rightarrow \tau'$ with simplification rules $\mathsf{ifeq}(l, l, e_1, e_2) \longrightarrow e_1$ and $\mathsf{ifeq}(l_1, l_2, e_1, e_2) \longrightarrow e_2$ when $l_1 \neq l_2$.

3.1 Computation

We define configurations $Id \in \mathsf{Conf}$ (and the auxiliary notions of store, evaluation context and computation redex) and the computation relation $Id \longmapsto Id' \mid ok$ (see Table 1).

– Stores $\mu \in \mathsf{S} \stackrel{\Delta}{=} \mathsf{L} \stackrel{fin}{\rightarrow} \mathsf{E}$ map locations to their contents.
– Evaluation contexts $\boxed{E \in \mathsf{EC} ::= \Box \mid E[do(\Box, [x]e)]}$.

- Configurations $(\mu, e, E) \in \mathsf{Conf} \stackrel{\Delta}{=} \mathsf{S} \times \mathsf{E} \times \mathsf{EC}$ consist of the current store μ, the program fragment e under consideration and its evaluation context E.
- Computation redexes $\boxed{r \in \mathsf{R} ::= do(e_1, [x]e_2) \mid ret(e) \mid new(e) \mid get(l) \mid set(l, e)}$.

When the program fragment under consideration is a computation redex, it enables a computation step with no need for further simplification (see Theorem 1).

Administrative steps, involve only the evaluation context

A.0 $(\mu, ret(e), \Box) \longmapsto ok$
A.1 $(\mu, ret(e_1), E[do(\Box, [x]e_2)]) \longmapsto (\mu, e_2[x := e_1], E)$
A.2 $(\mu, do(e_1, [x]e_2), E) \longmapsto (\mu, e_1, E[do(\Box, [x]e_2)])$

Imperative steps, involve only the store

I.1 $(\mu, new(e), E) \longmapsto (\mu\{l : e\}, ret(l), E)$, where $l \notin \mathsf{dom}(\mu)$
I.2 $(\mu, get(l), E) \longmapsto (\mu, ret(e), E)$, provided $e = \mu(l)$
I.3 $(\mu, set(l, e), E) \longmapsto (\mu\{l = e\}, ret(l), E)$, provided $l \in \mathsf{dom}(\mu)$

Table 1. Computation Relation for MML

The confluent simplification relation $\longrightarrow$ on terms extends in the obvious way to a confluent relation (denoted $\longrightarrow$) on stores, evaluation contexts, computation redexes and configurations.

A *complete program* corresponds to a closed term $e \in \mathsf{E}_0$ (with no occurrences of locations l), and its evaluation starts from the *initial configuration* $(\emptyset, e, \Box)$. The following properties ensure that only closed configurations are reachable (by $\longrightarrow$ and $\longmapsto$ steps) from initial ones.

Lemma 1.

1. *If* $(\mu, e, E) \longrightarrow (\mu', e', E')$, *then* $\mathsf{dom}(\mu') = \mathsf{dom}(\mu)$ *and* $\mathrm{FV}(\mu') \subseteq \mathrm{FV}(\mu)$, $\mathrm{FV}(e') \subseteq \mathrm{FV}(e)$ *and* $\mathrm{FV}(E') \subseteq \mathrm{FV}(E)$.
2. *If* $Id \longmapsto Id'$ *and* Id *is closed, then* Id' *is closed.*

When the program fragment under consideration is a computation redex, it does not matter whether simplification is done before or after computation.

Theorem 1 (Bisim). *If* $Id \equiv (\mu, e, E)$ *with* $e \in \mathsf{R}$ *and* $Id \stackrel{*}{\longrightarrow} Id'$, *then*

1. $Id \longmapsto D$ *implies* $\exists D'$ *s.t.* $Id' \longmapsto D'$ *and* $D \stackrel{*}{\longrightarrow} D'$
2. $Id' \longmapsto D'$ *implies* $\exists D$ *s.t.* $Id \longmapsto D$ *and* $D \stackrel{*}{\longrightarrow} D'$

where D *and* D' *range over* $\mathsf{Conf} \cup \{ok\}$.

Proof. An equivalent statement, but easier to prove, is obtained by replacing $\xrightarrow{*}$ with one-step parallel reduction. A key observation for proving the bisimulation result is that simplification applied to a computation redex r and an evaluation context E does not change the relevant structure (of r and E) for determining the computation step among those in Table 1.

3.2 Type Safety

We go through the proof of type safety. The result is standard and unsurprising, but we make some adjustments to the Subject Reduction (SR) and Progress properties, in order to stress the role of simplification $\longrightarrow$ and computation $\longmapsto$, when they are not bundled in one deterministic reduction strategy on configurations. First of all, we define well-formedness for configurations $\vdash_\Sigma Id$ and evaluation contexts $\Box\colon M\tau \vdash_\Sigma E\colon M\tau'$.

Definition 1. *We write* $\vdash_\Sigma (\mu, e, E) \overset{\Delta}{\Longleftrightarrow}$

- $\mathsf{dom}(\Sigma) = \mathsf{dom}(\mu)$
- $\mu(l) = e_l$ *and* $\Sigma(l) = \tau_l$ *imply* $\vdash_\Sigma e_l\colon \tau_l$
- *there exists* τ *such that* $\vdash_\Sigma e\colon M\tau$ *is derivable*
- *there exists* τ' *such that* $\Box\colon M\tau \vdash_\Sigma E\colon M\tau'$ *is derivable (see Table 2)*

$$\Box\ \frac{}{\Box\colon M\tau' \vdash_\Sigma \Box\colon M\tau'} \qquad do\ \frac{\Box\colon M\tau_2 \vdash_\Sigma E\colon M\tau' \quad \vdash_\Sigma [x]e\colon \tau_1 \Rightarrow M\tau_2}{\Box\colon M\tau_1 \vdash_\Sigma E[do(\Box, [x]e)]\colon M\tau'}$$

Table 2. Well-formed Evaluation Contexts for MML

Theorem 2 (SR).

1. *If* $\vdash_\Sigma Id_1$ *and* $Id_1 \longrightarrow Id_2$, *then* $\vdash_\Sigma Id_2$
2. *If* $\vdash_{\Sigma_1} Id_1$ *and* $Id_1 \longmapsto Id_2$, *then exists* $\Sigma_2 \supseteq \Sigma_1$ *s.t.* $\vdash_{\Sigma_2} Id_2$

Proof. The first claim is an easy consequence of Proposition 3. The second is proved by case-analysis on the computation rules of Table 1.

Theorem 3 (Progress). *If* $\vdash_\Sigma (\mu, e, E)$, *then one of the following holds*

1. $e \notin \mathsf{R}$ *and* $e \longrightarrow$, *or*
2. $e \in \mathsf{R}$ *and* $(\mu, e, E) \longmapsto$

Proof. When $e \in \mathsf{R}$ we have $(\mu, e, E) \longmapsto$, e.g. when e is $get(l)$ or $set(l, e')$, then $l \in \mathsf{dom}(\mu)$ by well-formedness of the configuration. When $e \notin \mathsf{R}$, then e cannot be a $\longrightarrow$-normal form, otherwise we get a contradiction with $\vdash_\Sigma e\colon M\tau$.

4 MMML: A Multi-stage Extension of MML

We describe a monadic metalanguage MMML obtained by adding staging to MML. At the level of syntax, type system and simplification the extension is *generic*, i.e. , applicable to any monadic metalanguage (as defined in Section 2).

- The BNF for types $\boxed{\tau \in \mathsf{T}+ = \langle\tau\rangle}$ is extended with code types.
- The BNF for term-constructors $\boxed{c \in \mathsf{C}+ = up \mid dn \mid c_V \mid c_M}$ is extended with *up*, *dn* and two recursive productions c_V and c_M, which capture the reflective nature of the extension (the set of term-constructors for MMML is infinite, although that for MML is finite). The type schema for the additional term-constructors are
  - $up\colon \tau \Rightarrow \langle\tau\rangle$ is MetaML cross-stage persistence (aka binary inclusion).
  - $dn\colon \langle\tau\rangle \Rightarrow M\tau$ is compilation of (potentially open) code. An attempt to compile open code causes an *unresolved link* error (an effect not present in MML), thus *dn* has a computational result type.
  - if $c\colon (\overline{\tau}_i \Rightarrow \tau_i | i \in m) \Rightarrow \tau$, then
    - $c_V\colon (\langle\overline{\tau}_i\rangle \Rightarrow \langle\tau_i\rangle | i \in m) \Rightarrow \langle\tau\rangle$ builds code representing a term $c(\ldots)$
    - $c_M\colon (\langle\overline{\tau}_i\rangle \Rightarrow M\langle\tau_i\rangle | i \in m) \Rightarrow M\langle\tau\rangle$ builds a computations that generates code representing $c(\ldots)$

    where $\langle\tau_i | i \in m\rangle$ stands for the sequence $(\langle\tau_i\rangle | i \in m)$. For instance, $\lambda_V\colon (\langle\tau_1\rangle \Rightarrow \langle\tau_2\rangle) \Rightarrow \langle\tau_1 \to \tau_2\rangle$ and $\lambda_M\colon (\langle\tau_1\rangle \Rightarrow M\langle\tau_2\rangle) \Rightarrow M\langle\tau_1 \to \tau_2\rangle$.

  The key difference between c_V and c_M (reflected in their type schema) is that generating code with c_M may have computational effects, while with c_V does not. For instance, the computation $\lambda_M([x]e)$ *generates a fresh name* (a new effect related to computation under a binder), performs the computation e to generate the code e' for the body of the λ-abstraction, and finally returns the code $\lambda_V([x]e')$ for the λ-abstraction.

The BNF for terms $e \in \mathsf{E}$ and the type system (for deriving judgments of the form $\Gamma \vdash_\Sigma e\colon \tau$) are extended in the only possible way, given the type schema for the term-constructors. In MMML (unlike $\lambda^{\bigcirc}$ and MetaML) there is no need to include level information in typing judgments, since it is already explicit in types and terms. For instance, a MetaML type τ at level 1 becomes $\langle\tau\rangle$ in MMML, and a λ at level 1 becomes a λ_V or λ_M.

Simplification $\longrightarrow$ is unchanged, i.e. , no new simplification rules are added. The properties of simplification established in Section 3 (i.e. , Proposition 1, 2 and 3) continue to hold and their proofs are unchanged.

Remark 3. One may wonder whether there is a need to have both c_M and c_V, or whether c_M can be defined in terms of c_V and term-constructors for computational types. Indeed, this is the case when c is **not a binder**. For instance, when $c\colon \tau_1 \Rightarrow \tau_2$ and $e\colon M\tau_1$, we could define $c_M(e)$ as $do(e, [x]ret(c_V(x)))$.

However, when c is **a binder**, like λ, one cannot move the computation of the body of $\lambda_M([x]e)$ outside the binder. One could adopt a more concrete representation of terms using first-order abstract syntax (FOAS), and introduce a

monadic operation *gensym*: $M\langle\tau\rangle$ to generate a fresh name (see [6]). But in this approach λ_V is no longer a binder, and this would be a drastic loss of abstraction for a reference semantics.

In λ-calculus one can encode a term-constructor c as a constant c' of higher-order type. For instance, $do(e_1, [x]e_2)$ becomes $do'@e_1@(\lambda x.e_2)$, where we adopt the standard infix notation for application @. Then we can use c'_V to encode c_M and c_V. For instance, $do_V(e_1, [x]e_2)$ becomes $do'_V@_V e_1@_V(\lambda_V x.e_2)$, and $do_M(e_1, [x]e_2)$ becomes $do@e_1@(\lambda c.do@(\lambda_M x.e_2)@(\lambda f.do'_V@_V c@_V f))$. With this encoding (unlike FOAS) there is no loss of abstraction, moreover it gives better control on code generation, e.g. $do_M(e_1, [x]e_2)$ (and its encoding) computes e_1 first, while $do@(\lambda_M x.e_2)@(\lambda f.do@e_1@(\lambda c.do'_V@_V c@_V f))$ computes e_2 first.

4.1 Computation

We now define configurations and the computation relation $Id \longmapsto Id' \mid ok \mid \mathsf{err}$ for MMML (see Table 3), where err indicates an unresolved link error at run-time. We must account for run-time errors, because we have adopted a permissive (and simple) type system. In the following we stress what auxiliary notions need to be changed when going from MML to MMML.

Remark 4. When adding staging, the modifications to the definition of $\longmapsto$ are fairly *modular*, but we cannot rely on a general theory (like rewrite rules for CRS) as in the case of simplification.

- Stores $\mu \in \mathsf{S} \stackrel{\Delta}{=} \mathsf{L} \stackrel{fin}{\to} \mathsf{E}$ are unchanged.
- Evaluation contexts $\boxed{E \in \mathsf{EC}+ = E[c_M(\overline{v}, [\overline{x}]\Box, \overline{f})]}$ are extended with one production, where $c \in \mathsf{C}$, $f ::= [\overline{x}]e$ is an abstraction, $v ::= [\overline{x}]ret(e)$ is a value abstraction. Moreover, $\overline{v}$, $\overline{x}$ and $\overline{f}$ must be consistent with the arity of c, for instance $E[\lambda_M([x]\Box)]$, $E[do_M(\Box, [x]e)]$ and $E[do_M(ret(e), [x]\Box)]$. Intuitively $E[\lambda_M([x]\Box)]$ says that the program fragment under consideration is generating code for the body of a λ-abstraction.
- A configuration $(X|\mu, e, E) \in \mathsf{Conf} \stackrel{\Delta}{=} \mathcal{P}_{fin}(\mathsf{X}) \times \mathsf{S} \times \mathsf{E} \times \mathsf{EC}$ has an additional component, i.e. , the set X of *names* generated so far. A name may leak outside the scope of its binder, thus X grows as the computation progresses.
- Computation redexes $\boxed{r \in \mathsf{R} += c_M(\overline{f}) \mid dn(vc)}$ are extended with two productions, where $c \in \mathsf{C}$, $\overline{f}$ must be consistent with the arity of c, and $\boxed{vc \in \mathsf{VC} ::= x \mid up(e) \mid c_V([\overline{x}_i]vc_i | i \in m)}$ is a code value. The redex $c_M(\overline{f})$ may generate fresh names, while $dn(vc)$ may cause an unresolved link error.

Compilation dn takes a code value vc of type $\langle\tau\rangle$ and computes the term e of type τ represented by vc (or fails if e does not exist). The represented term e is given by an operation similar to MetaML's demotion.

Definition 2 (Demotion). *The partial function* $\_\downarrow$ *mapping* $vc \in \mathsf{VC}$ *to the represented term is given by*

– $x{\downarrow}$ *is undefined;* $up(e){\downarrow} = e$ *(this is a base case, like* x*);*
$c_V([\overline{x}_i]vc_i|i \in m){\downarrow} = c([\overline{x}_i]e_i|i \in m)$ *when* $e_i = vc_i[\overline{x}_i := up(\overline{x}_i)]{\downarrow}$ *for* $i \in m$

where $up(\overline{x})$ *is the sequence* $(up(x_i)|i \in m)$ *when* $\overline{x} = (x_i|i \in m)$

Administrative and Imperative steps are as in Table 1, they do not modify the set X. Code generation steps, involve only the set X and the evaluation context

G.0 $(X|\mu, c_M, E) \longmapsto (X|\mu, ret(c_V), E)$ when the arity of c is ()
G.1 $(X|\mu, c_M([\overline{x}]e, \overline{f}), E) \longmapsto (X, \overline{x}|\mu, e, E[c_M([\overline{x}]\Box, \overline{f})])$ with $\overline{x}$ **renamed to avoid clashes** with X. In particular $(X|\mu, \lambda_M([x]e), E) \longmapsto (X, x|\mu, e, E[\lambda_M([x]\Box)])$
G.2 $(X|\mu, ret(e), E[c_M(\overline{v}, [\overline{x}]\Box)]) \longmapsto (X|\mu, ret(c_V(\overline{f}, [\overline{x}]e)), E)$ where $\overline{v} = ([\overline{x}_i]ret(e_i)|i \in m)$ and $\overline{f} = ([\overline{x}_i]e_i|i \in m)$. The free occurrences of $\overline{x}$ in e **get captured** by c_V, e.g. $(X|\mu, ret(e), E[\lambda_M([x]\Box)]) \longmapsto (X|\mu, ret(\lambda_V([x]e)), E)$
G.3 $(X|\mu, ret(e_1), E[c_M(\overline{v}, [\overline{x}_1]\Box, [\overline{x}_2]e_2, \overline{f})]) \longmapsto$
$(X, \overline{x}_2|\mu, e_2, E[c_M(\overline{v}, [\overline{x}_1]ret(e_1), [\overline{x}_2]\Box, \overline{f})])$ with $\overline{x}_2$ **renamed to avoid clashes** with X, and the free occurrences of $\overline{x}_1$ in e_1 **captured** by c_M.

Compilation step, may cause a run-time error

C.1 $(X|\mu, dn(vc), E) \longmapsto \begin{cases} (X|\mu, ret(e), E) & \text{if } e = vc{\downarrow} \\ \mathsf{err} & \text{if } vc{\downarrow} \text{ undefined} \end{cases}$

Table 3. Computation Relation for MMML

In an evaluation context for MMML, e.g. $E[\lambda_M([x]\Box)]$, the hole $\Box$ can be within the scope of a binder, thus an evaluation context E has not only a set of free variables, but also a sequence of captured variables.

Definition 3. *The sequence* $\mathrm{CV}(E)$ *of* captured variables *and the set* $\mathrm{FV}(E)$ *of* free variables *are defined by induction on the structure of* E

– $\mathrm{CV}(\Box) \triangleq \emptyset$ $\quad \mathrm{CV}(E[do(\Box, [x]e)]) \triangleq \mathrm{CV}(E)$
$\mathrm{CV}(E[c_M(\overline{v}, [\overline{x}]\Box, \overline{f})]) \triangleq \mathrm{CV}(E), \overline{x}$ *in particular* $\mathrm{CV}(E[\lambda_M([x]\Box)]) \triangleq \mathrm{CV}(E), x$
– $\mathrm{FV}(\Box) \triangleq \emptyset$ $\quad \mathrm{FV}(E[do(\Box, [x]e)]) \triangleq \mathrm{FV}(E) \cup (\mathrm{FV}([x]e) - \mathrm{CV}(E))$
$\mathrm{FV}(E[c_M(\overline{v}, [\overline{x}]\Box, \overline{f})]) \triangleq \mathrm{FV}(E) \cup (\mathrm{FV}(\overline{v}, \overline{f}) - \mathrm{CV}(E))$

As in the case of MML, the confluent simplification relation on terms extends to a confluent relation on the other syntactic categories. Also for MMML we can prove that only closed configurations are reachable from an initial one $(\emptyset|\emptyset, e, \Box)$, where $e \in \mathsf{E}_0$. However, the second clause of Lemma 2 is more subtle, in particular it ensures that $\mathrm{FV}(E)$ and $\mathrm{CV}(E)$ remain disjoint.

Lemma 2.

1. *If* $(X|\mu, e, E) \longrightarrow (X'|\mu', e', E')$, *then* $X' = X$, $\mathsf{dom}(\mu') = \mathsf{dom}(\mu)$, $\mathrm{CV}(E') = \mathrm{CV}(E)$, $\mathrm{FV}(\mu') \subseteq \mathrm{FV}(\mu)$, $\mathrm{FV}(e') \subseteq \mathrm{FV}(e)$ *and* $\mathrm{FV}(E') \subseteq \mathrm{FV}(E)$.

2. *If* $(X|\mu, e, E) \longmapsto\!\!\!\!\!\rightarrow (X'|\mu', e', E')$, $\mathrm{FV}(\mu, e) \cup \mathrm{CV}(E) \subseteq X$ *and* $\mathrm{FV}(E) \subseteq X - \mathrm{CV}(E)$, *then* $X \subseteq X'$, $dom(\mu) \subseteq dom(\mu')$, $\mathrm{FV}(\mu', e') \cup \mathrm{CV}(E') \subseteq X'$ *and* $\mathrm{FV}(E') \subseteq X' - \mathrm{CV}(E')$.

The bisimulation result (Theorem 1) is basically unchanged, but the proof must cover additional cases corresponding to the computation rules in Table 3.

4.2 Type Safety

In MMML the definitions of well-formed configuration $\Delta \vdash_\Sigma Id$ and evaluation context $\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'$ must take into account the set X. For this reason we need a type assignment Δ which maps names $x \in X$ to code types $\langle\tau\rangle$.

Definition 4. *We write* $\Delta \vdash_\Sigma (X|\mu, e, E) \overset{\Delta}{\Longleftrightarrow}$

- $dom(\Sigma) = dom(\mu)$ *and* $dom(\Delta) = X$
- $\mu(l) = e_l$ *and* $\Sigma(l) = \tau_l$ *imply* $\Delta \vdash_\Sigma e_l\colon \tau_l$
- *there exists* τ *such that* $\Delta \vdash_\Sigma e\colon M\tau$ *is derivable*
- *there exists* τ' *such that* $\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'$ *is derivable (see Table 4).*

$$\Box\ \frac{}{\Delta, \Box\colon M\tau' \vdash_\Sigma \Box\colon M\tau'} \qquad do\ \frac{\Delta, \Box\colon M\tau_2 \vdash_\Sigma E\colon M\tau' \quad \Delta \vdash_\Sigma [x]e\colon \tau_1 \Rightarrow M\tau_2}{\Delta, \Box\colon M\tau_1 \vdash_\Sigma E[do(\Box, [x]e)]\colon M\tau'}$$

$$c_M\ \frac{\begin{array}{c}\Delta, \Box\colon M\langle\tau\rangle \vdash_\Sigma E\colon M\tau' \\ \{\Delta \vdash_\Sigma v_i\colon \langle\overline{\tau}_i\rangle \Rightarrow M\langle\tau_i\rangle \mid i \in m\} \quad \{\Delta \vdash_\Sigma f_i\colon \langle\overline{\tau}_{m+1+i}\rangle \Rightarrow M\langle\tau_{m+1+i}\rangle \mid i \in n\}\end{array}}{\Delta, \{x_k\colon \langle\tau'_k\rangle | k \in p\}, \Box\colon M\langle\tau_m\rangle \vdash_\Sigma E[c_M(\overline{v}, [\overline{x}]\Box, \overline{f})]\colon M\tau'}\ \text{cond}$$

where the side-condition (cond) is: $\overline{v} = (v_i | i \in m)$ and $\overline{f} = (f_i | i \in n)$; $c_M\colon (\langle\overline{\tau}_i\rangle \Rightarrow M\langle\tau_i\rangle | i \in m + 1 + n) \Rightarrow M\langle\tau\rangle$; $\overline{\tau}_m = (\tau'_k | k \in p)$ and $\overline{x} = (x_k | k \in p)$

$$\text{in particular}\ \ \lambda_M\ \frac{\Delta, \Box\colon M\langle\tau_1 \to \tau_2\rangle \vdash_\Sigma E\colon M\tau'}{\Delta, x\colon \langle\tau_1\rangle, \Box\colon M\langle\tau_2\rangle \vdash_\Sigma E[\lambda_M([x]\Box)]\colon M\tau'}$$

Table 4. Well-formed Evaluation Contexts for MMML

Remark 5. The formation rule (c_M) for an evaluation context $E[c_M(\overline{v}, [\overline{x}]\Box, \overline{f})]$ says that the captured variables $\overline{x}$ must have a code type (this is consistent with the code generation rules (G.1) and (G.3) of Table 3) and that they should not occur free in E, $\overline{v}$ or $\overline{f}$ (this is consistent with the second property in Lemma 2).

Lemma 3. *If* $\Gamma \vdash_\Sigma vc\colon \langle\tau\rangle$ *and* $e = vc\downarrow$, *then* $\Gamma \vdash_\Sigma e\colon \tau$.

We can now formulate the SR and progress properties for MMML.

Theorem 4 (SR).

1. *If* $\Delta \vdash_\Sigma Id_1$ *and* $Id_1 \longrightarrow Id_2$*, then* $\Delta \vdash_\Sigma Id_2$
2. *If* $\Delta_1 \vdash_{\Sigma_1} Id_1$ *and* $Id_1 \longmapsto Id_2$*, then exist* $\Sigma_2 \supseteq \Sigma_1$ *and* $\Delta_2 \supseteq \Delta_1$ *s.t.* $\Delta_2 \vdash_{\Sigma_2} Id_2$

Proof. The first claim is straightforward (see Theorem 2). The second is proved by case-analysis on the computation rules, so we must cover the additional cases for the computation rules in Table 3, e.g.

(G.1) if Id_1 is $(X|\mu, \lambda_M([x]e), E)$, then Id_2 is $(X, x|\mu, e, E[\lambda_M([x]\Box)])$ and the typings $\Delta_1, x\colon\langle\tau_1\rangle \vdash_{\Sigma_1} e\colon M\langle\tau_2\rangle$ and $\Delta_1, \Box\colon M\langle\tau_1 \to \tau_2\rangle \vdash_{\Sigma_1} E\colon\tau'$ are derivable. Therefore we can take $\Sigma_2 \equiv \Sigma_1$ and $\Delta_2 \equiv \Delta_1, x\colon\langle\tau_1\rangle$.

(C.1) if Id_1 is $(X|\mu, dn(vc), E)$, then Id_2 is $(X, x|\mu, ret(e), E)$ with $e = vc\downarrow$ and the typings $\Delta_1 \vdash_{\Sigma_1} vc\colon\langle\tau\rangle$ and $\Delta_1, \Box\colon M\tau \vdash_{\Sigma_1} E\colon\tau'$ are derivable. By Lemma 3 $\Delta_1 \vdash_{\Sigma_1} e\colon\tau$ is derivable, therefore we can take $\Sigma_2 \equiv \Sigma_1$ and $\Delta_2 \equiv \Delta_1$.

Lemma 4. *If* $\Delta \vdash_\Sigma e\colon\tau$ *and* e *is a* $\longrightarrow$*-normal form, then*

- $\tau \equiv \mathsf{nat}$ *implies* e *is a natural number*
- $\tau \equiv M\tau$ *implies* e *is a computation redex*
- $\tau \equiv \mathsf{ref}\ \tau$ *implies* e *is a location*
- $\tau \equiv \langle\tau'\rangle$ *implies* e *is a code value*
- $\tau \equiv (\tau_1 \to \tau_2)$ *implies* e *is a* λ*-abstraction*

Proof. By induction on the derivation of $\Delta \vdash_\Sigma e\colon\tau$. The base cases are: x, *up*, l, λ, *ret*, *do*, *new* and c_M. The inductive steps are: *get*, *set*, *dn*, c_V and @ (@ is impossible because by the IH one would have a β-redex).

Theorem 5 (Progress). *If* $\Delta \vdash_\Sigma (X|\mu, e, E)$*, then one of the following holds*

1. $e \notin \mathsf{R}$ *and* $e \longrightarrow$*, or*
2. $e \in \mathsf{R}$ *and* $(X|\mu, e, E) \longmapsto$

Proof. When $e \in \mathsf{R}$ we have $(\mu, e, E) \longmapsto$ (see Theorem 3). When $e \notin \mathsf{R}$, then e cannot be a $\longrightarrow$-normal form. otherwise we get a contradiction with $\Delta \vdash_\Sigma e\colon M\tau$ and Lemma 4.

5 Examples

We give simple examples of computations in MMML to illustrate subtle points of multi-stage programming. For readability, we use Haskell's do-notation $x \Leftarrow e_1; e_2$ (or $e_1; e_2$) for $do(e_1, [x]e_2)$ (when $x \notin \mathrm{FV}(e_2)$) and write $\lambda_B x.e$ for $\lambda_B([x]e)$.

Scope extrusion: a bound variable x leaks in the store.

$$l \Leftarrow new(0_V); (\lambda_M x.set(l,x); ret(x)) : M\langle \mathsf{nat} \to \mathsf{nat} \rangle$$

1. $(\emptyset \mid \emptyset, \quad new(0_V), \quad l \Leftarrow \Box; \lambda_M x.set(l,x); ret(x))$ create a location l
2. $(\emptyset \mid l = 0_V, \quad \lambda_M x.set(l,x); ret(x), \quad \Box)$ generate a fresh name x
3. $(x \mid l = 0_V, \quad set(l,x), \quad \lambda_M x.\Box; ret(x))$ assign x to l
4. $(x \mid l = x, \quad ret(x), \quad \lambda_M x.\Box)$ complete code generation of λ-abstraction
5. $(x \mid l = x, \quad ret(\lambda_V x.x), \quad \Box)$ x is bound by λ_V, but a copy is also left in the store.

The semantics in [3] is more conservative, when a variable leaks in the store it is bound by *dead-code annotation.*, but on closed values the two semantics *agree.*

Recapturing of extruded variable by its binder.

$$\lambda_M x.\ l \Leftarrow new(x); get(l) : M\langle \tau \to \tau \rangle$$

1. $(\emptyset \mid \emptyset, \quad \lambda_M x.\ l \Leftarrow new(x); get(l), \quad \Box)$ generate x, then create l
2. $(x \mid l = x, \quad get(l), \quad \lambda_M x.\Box)$ get content of l
3. $(x \mid l = x, \quad ret(x), \quad \lambda_M x.\Box)$ complete code generation of λ-abstraction
4. $(x \mid l = x, \quad ret(\lambda_V x.x), \quad \Box)$ x is bound by λ_V.

This form of recapturing is allowed by [17], but not by [3].

No recapturing of extruded variable by another binder using the *same* name.

$$l \Leftarrow new(0_V); (\lambda_M x.set(l,x); ret(x)); z \Leftarrow get(l); ret(\lambda_V x.z) : M\langle \tau \to \mathsf{nat} \rangle$$

1. $(\emptyset \mid l = 0_V, \quad (\lambda_M x.set(l,x); ret(x)), \quad \Box; z \Leftarrow get(l); ret(\lambda_V x.z))$
   generate x and assign it to l
2. $(x \mid l = x, \quad ret(\lambda_V x.x), \quad \Box; z \Leftarrow get(l); ret(\lambda_V x.z))$
   first code generation completed
3. $(x \mid l = x, \quad get(l), \quad z \Leftarrow \Box; ret(\lambda_V x.z))$ get content of l
4. $(x \mid l = x, \quad ret(x), \quad z \Leftarrow \Box; ret(\lambda_V x.z))$
   complete code generation of λ-abstraction
5. $(x \mid l = x, \quad ret(\lambda_V x'.x), \quad \Box)$
   the bound variable x is renamed by substitution $ret(\lambda_V x.z)[z := x]$

No recapturing of extruded variable by its binder after code generation.

$$\begin{aligned} l \Leftarrow new(0_V); z \Leftarrow (\lambda_M x.\lambda_M y.set(l,y); ret(x)); \\ f \Leftarrow dn(z); u \Leftarrow get(l); ret(f\ u) : M(\mathsf{nat} \to \langle \mathsf{nat} \rangle) \end{aligned}$$

1. $(x, y \mid l = y, \quad ret(\lambda_V x.\lambda_V y.x), \quad z \Leftarrow \Box; f \Leftarrow dn(z); u \Leftarrow get(l); ret(f\ u))$
   code generation completed, y is bound by λ_V and leaked in the store
2. $(x, y \mid l = y, \quad dn(\lambda_V x.\lambda_V y.x), \quad f \Leftarrow \Box; u \Leftarrow get(l); ret(f\ u))$ compile code

3. $(x, y \mid l = y, \quad ret(\lambda x.\lambda y.x), \quad f \Leftarrow \Box; u \Leftarrow get(l); ret(f\ u))$
   get content of l and apply f to it
4. $(x, y \mid l = y, \quad ret((\lambda x.\lambda y.x)\ y), \Box)$ the result simplifies to $(\lambda y'.y)$, because the bound variable y is renamed by β-reduction.

When y is recaptured by λ_V, it becomes a bound variable and can be renamed. Therefore, the connection with the (free) occurrences of y left in the store (or the program fragment under consideration) is lost.

6 Related Work and Discussion

We discuss related work and some issues specific to MMML. A more general discussion of open issues in meta-programming can be found in [14].

Comparison with MetaML, $\lambda^{\bigcirc}$ *and* $\lambda^{\mathfrak{M}}$. The motivation for looking at the interactions between computational effects and run-time code generation comes from MetaML [9, 15, 16, 3]. We borrow code types from MetaML (and $\lambda^{\bigcirc}$ of [4]), but use annotated term-constructors as in $\lambda^{\mathfrak{M}}$ of [4] (see also [7]), so that simplification and computation rules are *level insensitive*. Indeed, the term-constructors $c \in \mathsf{C}$ of MMML can be given by an alternative BNF

$$c ::= ret_B \mid do_B \mid \lambda_B \mid @_B \mid new_B \mid get_B \mid set_B \mid l_B \mid up_B \mid dn_B \quad \text{with } B \in \{V, M\}^*$$

For instance, λ_B is λ when B is empty; if c is λ_B, then c_V and c_M are given by λ_{BV} and λ_{BM} respectively. However, MMML's annotations are sequences $B \in \{V, M\}^*$, while those of $\lambda^{\mathfrak{M}}$ are natural number n. A sequence B identifies a natural number n, namely the length of B, moreover for each $i < n$ it says whether computation at that *level* has been completed, as expressed by the different typing for c_V and c_M. The refined annotations of term-constructors (and computational types) allow to distinguish the following situations:

- $(\lambda_M x.e, E)$ we start generating code for a λ-abstraction
- $(e, E[\lambda_M x.\Box])$ we have generated a fresh name x, and start generating code for the body
- $(e, E[\lambda_M x.E'])$ we are somewhere in the middle of the computation generating code for the body
- $(ret(e), E[\lambda_M x.\Box])$ we have the code for the body of the λ-abstraction
- $(ret(\lambda_V x.e), E)$ we have the code for the λ-abstraction

All operational semantics proposed for MetaML or $\lambda^{\bigcirc}$ do not make these fine-grain distinctions. Only [10], which extends $\lambda^{\Box}$ of [5] with names a la FreshML (and *intensional analysis*), has an operational semantics with steps modeling fresh name generation and recapturing, but its relations with $\lambda^{\bigcirc}$ and MetaML have not been investigated, yet.

The *up* and *dn* primitives of MMML are related to cross-stage persistence $\%e$ and code execution run e of MetaML. In MMML demotion $vc\downarrow$ is partial, thus

evaluation of $dn(vc)$ may raise an unresolved link error, while in MetaML demotion is total, and an unresolved link error is raised only when evaluating x (at level 0). However, in [3] demotion is applied only to closed values, during evaluation of well-typed programs.

Multi-lingual extensions. It is easy to extend a monadic metalanguage, like MMML, to cope with a variety of programming languages: each programming language PL_i is modeled by a different monad M_i with its own set of operations. However, one should continue to have **one code type** constructor $\langle\tau\rangle$, i.e. , the representation of terms should be uniform. Therefore, there should be one $up\colon\tau\Rightarrow\langle\tau\rangle$ and one c_V (for each c), but several $dn_i\colon\langle\tau\rangle\Rightarrow M_i\tau$ and c_{M_i}, one for each monad M_i. In this way, we could have terms of type $M_1\langle M_2\tau\rangle$, which correspond to a program written in PL_1 for generating programs written in PL_2.

Compilation strategies. The compilation step (C.1) in Table 3 uses the demotion operation of Definition 2, which returns the term $vc\downarrow$ of type τ represented by a code value vc of type $\langle\tau\rangle$ (if such a term exists). One could adopt a lazier compilation strategy, which delays the compilation of parts of the code. A lazy strategy has the effect of delaying unresolved link errors, including the possibility of never raising them (when part of the code is dead). For instance, a possible clause for *lazy demotion* is $ret_V(e)\downarrow = dn(e)$. A more aggressive approach is to replace the compilation step with simplification rules

$$dn(up(e)) \longrightarrow e \qquad dn(c_V([\overline{x}_i]e_i|i\in m)) \longrightarrow c([\overline{x}_i]dn(e_i[\overline{x}_i := up(\overline{x}_i)])|i\in m)$$

However, one must modify the type system to ensure the SR and progress property,but changing the type schema for dn to $\langle\tau\rangle\Rightarrow\tau$ is not enough!

Type systems. We have adopted a simple type system for MMML, which does not detect statically all run-time errors. In particular, we have not included the closed type constructor $[\tau]$ of MetaML for two reasons:

1. there are alternative approaches to prevent link errors *incomparable* with the closed type approach (e.g. the *region-based* approach of [17] and the *environment classifier* approach of [11])
2. it requires *dead-code annotations* $(x)e$ that are instrumental to the proof of type safety.

Better type systems are desirable not only for detecting errors statically, but also to provide more accurate type schema for dn, e.g. $dn\colon[\langle\tau\rangle]\Rightarrow\tau$, which could justify replacing the *compilation step* by local simplification rules (see above). [10] is the best attempt up-to-date in addressing typing issues, although it does not explicitly consider computational effects. The adaptation of Nanevski's type system to MMML, e.g. refining code types $\langle\tau|C\rangle$ with a set C of names, is a subject for further research. Also the type system of [11] (where one has several code type constructors $\langle\tau\rangle^\alpha$, corresponding to different ways of representing

terms) could be adapted to MMML, but at a preliminary check it seems that the more accurate type schema $(\forall\alpha.\langle\tau\rangle^{\alpha})\Rightarrow\forall\alpha.\tau$ for *dn* is insufficient to validate the local simplification rules for compilation.

Uniform representation in Logical Frameworks. The code types of MMML provide a uniform representation of terms, similar to the (weak) Higher-Order Abstract Syntax (HOAS) encoding of object logics in a logical framework (LF). Of course, in a LF there are stronger requirements on HOAS encodings, but any advance in the area of LF is likely to advance the state-of-the-art in metaprogramming. Recently [10] has made significant advances in the area of *intensional analysis*, i.e. , the ability to analyze code (see [14]), by building on [13].

Monadic intermediate languages. [1] advocates the use of MIL for expressing optimizing transformations. Also MMML could be used for this purpose, but for having non-trivial optimizations one has to introduce more aggressive simplifications (than those strictly needed for defining the operational semantics) and refine monadic types with effect information as done in [1]. In general, we expect β-conversion $@(\lambda([x]e_2), e_1) \approx e_2[x{:=}e_1]$ and the following equivalences to be *observationally sound*

- $do(ret(e_1), [x]e_2) \approx e_2[x{:=}e_1]$
- $c_M([\overline{x}_i]ret(e_i)|i \in m) \approx ret(c_V([\overline{x}_i]e_i|i \in m))$

while other equivalences, like $@_V(\lambda_V([x]e_2), e_1) \approx e_2[x{:=}e_1]$, are more fragile (e.g. they fail when the language is extended with intensional analysis).

Acknowledgments. We thank Francois Pottier, Amr Sabry, Walid Taha for useful discussions, and the anonymous referees for their valuable comments.

References

[1] N. Benton and A. Kennedy. Monads, effects and transformations. In *Proceedings of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS-99)*, volume 26 of *Electronic Notes in Theoretical Computer Science*, Paris, September 1999. Elsevier.

[2] G. Berry and G. Boudol. The chemical abstract machine. In *Conf. Record 17th ACM Symp. on Principles of Programmming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan. 1990*, pages 81–94. ACM Press, New York, 1990.

[3] C. Calcagno, E. Moggi, and T. Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, to appear.

[4] R. Davies. A temporal-logic approach to binding-time analysis. In *the Symposium on Logic in Computer Science (LICS '96)*, pages 184–195, New Brunswick, 1996. IEEE Computer Society Press.

[5] R. Davies and F. Pfenning. A modal analysis of staged computation. In *the Symposium on Principles of Programming Languages (POPL '96)*, pages 258–270, St. Petersburg Beach, 1996.

[6] A. Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Proc. of 5th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'01, Krakow, Poland, 2–5 May 2001*, volume 2044 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, Berlin, 2001.

[7] R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.

[8] J. W. Klop. *Combinatory Reduction Systems*. PhD thesis, University of Utrecht, 1980. Published as Mathematical Center Tract 129.

[9] The MetaML Home Page, 2000. Provides source code and documentation online at http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html.

[10] A. Nanevski. Meta-programming with names and necessity. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP-02)*, ACM SIGPLAN notices, New York, October 2002. ACM Press.

[11] M. F. Nielsen and W. Taha. Environment classifiers. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, N.Y., January 15–17 2003. ACM Press.

[12] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, and et. al. Haskell 1.4: A non-strict, purely functional language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, Mar 1997. World Wide Web version at `http://haskell.cs.yale.edu/haskell-report`.

[13] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Programme Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, 2000.

[14] T. Sheard. Accomplishments and research challenges in meta-programming. In W. Taha, editor, *Proc. of the Int. Work. on Semantics, Applications, and Implementations of Program Generation (SAIG)*, volume 2196 of *LNCS*, pages 2–46. Springer-Verlag, 2001.

[15] W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from ftp://cse.ogi.edu/pub/tech-reports/README.html.

[16] W. Taha and T. Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000.

[17] P. Thiemann and D. Dussart. Partial evaluation for higher-order languages with state, 1999. Available from `http://www.informatik.uni-freiburg.de/~thiemann/papers/index.html`

[18] P. Wadler. The marriage of effects and monads. In *the International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 63–74. ACM, June 1999.

[19] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Multi-level Meta-reasoning with Higher-Order Abstract Syntax

Alberto Momigliano and Simon J. Ambler

Department of Mathematics and Computer Science, University of Leicester,
Leicester, LE1 7RH, U.K.
A.Momigliano@mcs.le.ac.uk, S.Ambler@mcs.le.ac.uk

Abstract. Combining Higher Order Abstract Syntax (HOAS) and (co)-induction is well known to be problematic. In previous work [1] we have described the implementation of a tool called Hybrid, within Isabelle HOL, which allows object logics to be represented using HOAS, and reasoned about using tactical theorem proving and principles of (co)induction. Moreover, it is definitional, which guarantees consistency within a classical type theory. In this paper we describe how to use it in a multi-level reasoning fashion, similar in spirit to other meta-logics such $FO\lambda^{\Delta IN}$ and *Twelf*. By explicitly referencing provability, we solve the problem of reasoning by (co)induction in presence of non-stratifiable hypothetical judgments, which allow very elegant and succinct specifications. We demonstrate the method by formally verifying the correctness of a compiler for (a fragment) of Mini-ML, following [10]. To further exhibit the flexibility of our system, we modify the target language with a notion of non-well-founded closure, inspired by Milner & Tofte [16] and formally verify via co-induction a subject reduction theorem for this modified language.

1 Introduction

Higher Order Abstract Syntax (HOAS) is a representation technique, dating back to Church, where binding constructs in an object logic are encoded within the function space provided by a meta-language based on a λ-calculus. This specification is generic enough that sometimes HOAS has been identified [12] merely with a mechanism to delegate α-conversion to the meta-language. While this is important, it is by no means the whole story. As made clear by the LF and λProlog experience, further benefits come when, for an object logic type i, this function space is taken to be $\mathsf{i} \Rightarrow \mathsf{i}$, or a subset of it. Then object-logic substitution can be rendered as meta-level β-conversion. A meta-language offering this facility is a step up, but there is still room for improvement. Experiments such as the one reported in [17] suggest that the full benefits of HOAS can only be enjoyed when it is paired with hypothetical and parametric judgments. A standard example is the full HOAS encoding of type-inference in which the type environment of a term is captured abstractly without any reference to a list of variable/type pairs. Though this is appealing, such judgments are generally not

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 375–391, 2003.

inductive since they may contain *negative* occurrence of the predicate being defined. This raises the question of how we are going to reason on such encodings, in particular, are there induction and case analysis principles available?

A solution that has emerged in the last five years is that *specification* and (inductive) *meta-reasoning* should be dealt with in a single system but at different *levels*. One such meta-logic is Miller & McDowell $FO\lambda^{\Delta IN}$ [14]; it is is based on higher-order intuitionistic logic augmented with definitional reflection [8] — to provide support for case analysis — and induction on natural numbers. Consistency is ensured by cut-elimination. Inside this meta-language they develop a *specification logic* (SL) which in turn it is used to specify the *object-logic* (OL) under study. This partition solves the problem of meta-reasoning in the presence of negative occurrences, since hypothetical judgments are now encapsulated within the OL and therefore not required to be inductive. The price to pay is this additional layer where we explicitly reference provability and the necessity therefore of a sort of meta-interpreter (the SL logic) to access it.

Very recently, Felty [6] has suggested that, rather than implementing an interactive theorem prover for $FO\lambda^{\Delta IN}$ from scratch, the latter can be simulated within an existing system (Coq in that case); in particular, definitional reflection is mimicked by the elimination rules of inductive types. Nevertheless, this is not quite enough, as reasoning by inversion crucially depends on simplifying in the presence of constructors. Since some of the higher-order ones may be non-inductive, Felty recurs to the assumption of a set of axioms stating the freeness and extensionality properties of constructors in the given signature. Under those conditions the author shows, in the example of the formalization of subject reduction for Mini-ML, how it is possible to replicate, in an well-understood and interactive setting, the style of proofs typical of $FO\lambda^{\Delta IN}$; namely the result is proven without "technical" lemmas foreign to the mathematics of the problem.

In previous work [1] we have described the implementation of a higher-order meta-language, called Hybrid, within Isabelle HOL, which provides a form of HOAS for the user to represent object logics. The user level is separated from the infrastructure, in which HOAS is implemented *definitionally* via a de Bruijn style encoding.

In this paper we adopt most of Felty's architecture, with the notable difference of using Hybrid rather than Coq as the basic meta-language. A graphical depiction of the proposed architecture is in Figure 1. Moreover, we take a further departure in design: we suggest to push at the OL *only* those judgments which would not be inductive, and to leave the rest at the Isabelle HOL level. We claim that this framework has several advantages:

- The system is more trustworthy: freeness of constructors and more importantly extensionality properties at higher types are not assumed, but proven via the related properties of the infrastructure, see Subsection 3.1.
- The mixing of meta-level and OL judgments makes proofs more easily mechanizable and allows us to use co-induction which is still unaccounted for in a logic such as $FO\lambda^{\Delta IN}$.

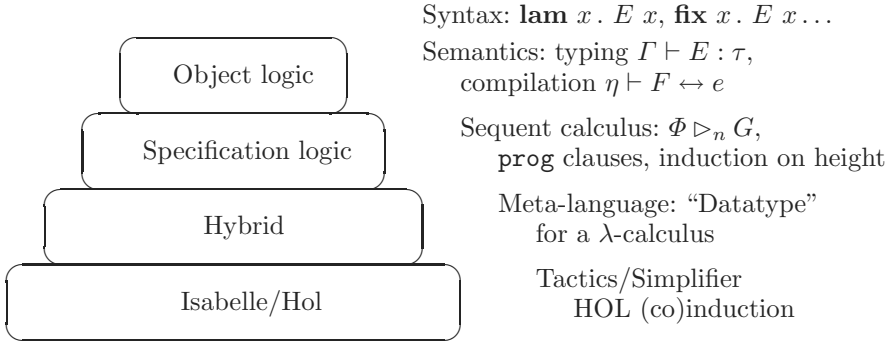


Fig. 1. Multi-Level Architecture

– More in general, there is a fruitful interaction between (co)-induction principles, Isabelle HOL datatypes, classical reasoning and hypothetical judgments, which tends to yield more automation than in a system such as Coq.

Our approach is also comparable with *Twelf* [21] (see Section 6), but is has a low mathematical overhead, as it simply consists of a package on top of Isabelle HOL. In a sense, we could look at Hybrid as a way to "compile" HOAS meta-proofs, such as Twelf's, into the well-understood setting of higher-order logic.

We demonstrate the method by formally verifying the correctness of (part of) a compiler for a fragment of Mini-ML, following [10]. To further exhibit the flexibility of our system, we modify the target language with a notion of non-well-founded closure, inspired by Milner & Tofte's paper [16] and formally verify, via co-induction, a subject reduction theorem for this new language.

The paper is organized as follows: in the next Section 2 we introduce at the informal level the syntax and semantics of our case study. Section 3 recalls some basics notion of Hybrid and its syntax-representing techniques. In Section 4 we introduce the multi-level architecture, while Section 5 is the heart of the paper, detailing the formal verification of compiler correctness. We conclude with a few words on related and future work (Section 6 and 7).

We use a pretty-printed version of Isabelle HOL concrete syntax; a rule (a sequent) with conclusion C and premises $H_1 \dots H_n$ will be represented as $[\![H_1; \dots; H_n]\!] \Longrightarrow C$. An Isabelle HOL type declaration has the form $s :: [t_1, \dots t_n] \Rightarrow t$. Isabelle HOL connectives are represented via the usual logical notation. Free variables (upper-case) are implicitly universally quantified. The sign "$\equiv$" (Isabelle meta-equality) is used for *equality by definition*, $\bigwedge$ for Isabelle universal meta-quantification. The keyword **MC-Theorem** denotes a machine-checked theorem, while *(Co)Inductive* introduces a (co)inductive relation in Isabelle HOL. We have tried to use the same notation for mathematical and formalized judgments. To facilitate the comparison with the Hannan & Pfenning's approach, we have used the same nomenclature and convention as [18].

2 The Case Study

Compiler verification can be a daunting task, but it is a good candidate for mechanization. Restricting to functional languages (see [11] for issue related to compilation of Java for example), some first attempts were based on denotational semantics [3, 13]. Using instead operational semantics has the advantage that the meaning of both high and low level languages can be expressed in a common setting. In particular, operational semantics and program translations can be represented using deductive systems. This approach started in [4][1]. Other papers, for example [9], have explored aspects of higher-order encoding of abstract machines and compilation and [10] contains the first attempt to carry formal verification of compiler correctness in LF. Only recently the notion of *regular world* introduced in [21] provides a satisfying foundation for the theory of *mode, termination and coverage* checking [19], which has been developed to justify using LF for meta-reasoning.

We follow the stepwise approach to compilation in [10]; however, for reason of space, we limit ourselves to the verification of the correspondence between the big-step semantics of a higher-order functional language and the evaluation to closure of its translation into a target language of first-order expressions. For interest, though, we add a co-inductive twist due to Milner & Tofte [16] in the treatment of fix points, which is analogous to the semantics of *letrec* in the original formulation of Mini-ML.

The language we utilize here is a strict lambda calculus augmented with a fix point operator, although it could be easily generalized to the rendition of Mini-ML in [18]. This fragment is sufficient to illustrate the main ideas without cluttering the presentation with too many details. The types and terms of the source language are given respectively by:

$$\begin{array}{l} \textit{Types}\ \tau ::= \mathbf{i} \mid \tau_1\ \mathsf{arrow}\ \tau_2 \\ \textit{Terms}\ e ::= x \mid \mathbf{lam}\ x\,.\ e \mid e\ @\ e' \mid \mathbf{fix}\ x\,.\ e \end{array}$$

The rules for call-by-value operational semantics ($e \Downarrow v$) and type inference ($\Gamma \vdash e : \tau$) are standard and are omitted — see Subsection 4.2 for the implementation. The usual subject reduction for this source language holds.

2.1 Compilation

We start the compilation process by translating the higher-order syntax into a first-order one. Terms are compiled into a simple calculus with explicit substitutions, where de Bruijn indexes are represented by appropriate shifting on the 1 numeral:

$$dB\ \textit{Terms}\ F ::= 1 \mid F\uparrow \mid \mathbf{lam}'\ F \mid F\ @'\ F' \mid \mathbf{fix}'\ F$$

[1] A partial verification via first-order encoding can be found in [2].

Then we introduce environments and closures:

$$\begin{array}{c} \textit{Environments} \quad \eta ::= \cdot \mid \eta; W \mid \eta + F \\ \textit{Values } W ::= \{\eta, F\} \end{array}$$

In this setting the only possible values are *closures*, yet the presence of fix-points requires environments to possibly contain unevaluated expressions, via the environment constructor '+'. We will see in Subsection 2.2 how this can be refined using a notion of *non-well-founded* closure.

The operational semantics of this language (judgment $\eta \vdash F \hookrightarrow W$) uses environments to represent mappings of free variables to values and closures for terms whose free variables have values in the environment. We remark that due to the presence of unrestricted fix points, the rule $\mathbf{fev\_1}^+$ in not just a look-up, but requires the evaluation of the body of the fix point.

$$\frac{}{\eta; W \vdash 1 \hookrightarrow W}\,\mathbf{fev\_1} \qquad \frac{\eta \vdash F \hookrightarrow W}{\eta; W' \vdash F \uparrow \hookrightarrow W}\,\mathbf{fev\_}\uparrow$$

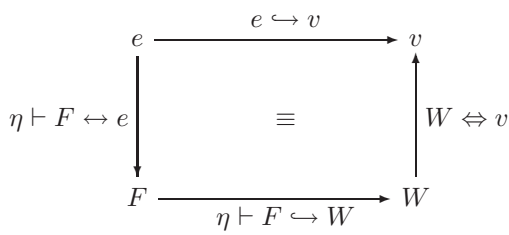
$$\frac{\eta \vdash F \hookrightarrow W}{\eta + F \vdash 1 \hookrightarrow W}\,\mathbf{fev\_1}^+ \qquad \frac{\eta \vdash F \hookrightarrow W}{\eta + F' \vdash F \uparrow \hookrightarrow W}\,\mathbf{fev\_}\uparrow^+$$

$$\frac{}{\eta \vdash \mathbf{lam}'\ F \hookrightarrow \{\eta, \mathbf{lam}'\ F\}}\,\mathbf{fev\_lam}' \qquad \frac{\eta + \mathbf{fix}'\quad F \vdash F \hookrightarrow W}{\eta \vdash \mathbf{fix}'\quad F \hookrightarrow W}\,\mathbf{fev\_fix}'$$

$$\frac{\eta \vdash F_1 \hookrightarrow \{\eta', \mathbf{lam}'\ F_1'\} \qquad \eta \vdash F_2 \hookrightarrow W_2 \qquad \eta'; W_2 \vdash F_1' \hookrightarrow W}{\eta \vdash F_1\ @'\ F_2 \hookrightarrow W}\,\mathbf{fev\_@}'$$

The judgment $\eta \vdash F \leftrightarrow e$ (Figure 2) elegantly accomplishes the translation to the dB language using parametric and hypothetical judgments for the binding constructs: for the fix point, we assume we have extended the environment with a new *expression* parameter f; in the function case, the parameter w ranges over *values* and the judgment is mutually recursive with value translation, $W \Leftrightarrow v$. As remarked in [18], this translation is total, but can be non-deterministic, when η and e are given and F is computed.

The verification of compiler correctness can be graphically represented as:



We discuss the statement and the proof once we provide the mechanization in Section 5.

$$\frac{\dfrac{}{w \Leftrightarrow x}\,u \quad \vdots \quad \eta; w \vdash F \leftrightarrow e}{\eta \vdash \mathbf{lam}'\ F \leftrightarrow \mathbf{lam}\ x\,.\,e}\,\mathbf{tr\_lam}^{w,x,u} \qquad \frac{\dfrac{}{\eta \vdash f \leftrightarrow x}\,u \quad \vdots \quad \eta + f \vdash F \leftrightarrow e}{\eta \vdash \mathbf{fix}'\ F \leftrightarrow \mathbf{fix}\ x\,.\,e}\,\mathbf{tr\_fix}^{f,x,u}$$

$$\frac{W \Leftrightarrow e}{\eta; W \vdash 1 \leftrightarrow e}\,\mathbf{tr\_1} \qquad \frac{\eta \vdash F \leftrightarrow e}{\eta; W \vdash F\!\uparrow \leftrightarrow e}\,\mathbf{tr\_}\uparrow$$

$$\frac{\eta \vdash F \leftrightarrow E}{\eta + F \vdash 1 \leftrightarrow e}\,\mathbf{tr\_1}^{+} \qquad \frac{\eta \vdash F \leftrightarrow e}{\eta + F' \vdash F\!\uparrow \leftrightarrow e}\,\mathbf{tr\_}\uparrow^{+}$$

$$\frac{\eta \vdash F_1 \leftrightarrow e_1 \qquad \eta \vdash F_2 \leftrightarrow e_2}{\eta \vdash F_1\ @'\ F_2 \leftrightarrow e_1 @\ e_2}\,\mathbf{tr\_app} \qquad \frac{\eta \vdash \mathbf{lam}'\ F \leftrightarrow \mathbf{lam}\ x\,.\,e}{\{\eta, \mathbf{lam}'\ F\} \Leftrightarrow \mathbf{lam}\ x\,.\,e}\,\mathbf{vtr\_lam}$$

Fig. 2. Translation to modified dB terms and values

2.2 Compilation via Non-Well-Founded Closures

In order to illustrate the flexibility of our approach we now depart from [18] and draw inspiration from [16] to give a different and simplified treatment of fix points. First, we take up the SML-like restriction that the body of a fix point is always a lambda. Then the idea is simply to allow a fix point to evaluate to a "circular" closure, where, intuitively, the free variable points to the closure itself. This means swapping the following rule for $\mathbf{fev\_fix}'$ and dropping rules $\mathbf{fev\_1}^{+}$ and $\mathbf{fev\_}\uparrow^{+}$. The analogous rules in the translation are also dropped.

$$\frac{cl = \{(\eta; cl), \mathbf{lam}'\ F\}}{\eta \vdash \mathbf{fix}'\ (\mathbf{lam}'\ F) \hookrightarrow cl}\,\mathbf{fev\_fix}'\ ^{*}$$

This amounts to a *non-well-founded* notion of closure. Other versions are possible, viz. recursive environments, see [4] for a discussion.

To exemplify the style of co-inductive reasoning entailed, we adapt the proof of subject reduction for closures in [16]. First we introduce typing for dB terms:

$$\frac{}{\Delta; \tau \vdash 1 : \tau}\,\mathbf{ftp\_1} \qquad \frac{\Delta \vdash F : \tau}{\Delta; \tau' \vdash F\!\uparrow : \tau}\,\mathbf{ftp\_}\uparrow \qquad \frac{\Delta; \tau \vdash F : \tau'}{\Delta \vdash \mathbf{lam}'\ F : \tau\ \mathsf{arrow}\ \tau'}\,\mathbf{ftp\_lam}'$$

$$\frac{\Delta; \tau \vdash F : \tau}{\Delta \vdash \mathbf{fix}'\ F : \tau}\,\mathbf{ftp\_fix}' \qquad \frac{\Delta \vdash F_1 : \tau'\ \mathsf{arrow}\ \tau \qquad \Delta \vdash F_2 : \tau'}{\Delta \vdash F_1\ @'\ F_2 : \tau}\,\mathbf{ftp\_@}'$$

Typing of closures can be seen as taking the greatest fix point of the following rules, where we refer the reader to *ibid.* for motivation and examples:

$$\frac{\Delta \vdash \mathbf{lam}'\ F : \tau \qquad \eta :' \Delta}{\{\eta, \mathbf{lam}'\ F\} : \tau}\ \mathbf{tpc} \qquad \frac{}{\cdot :' \cdot}\ \mathbf{etp}\cdot \qquad \frac{W : \tau \qquad \eta :' \Delta}{(\eta; W) :' (\Delta; \tau)}\ \mathbf{etp;}$$

We note the "circularity" of rule **tpc** and **etp;**, where the former requires well-typedness of value environment w.r.t. type environment.

Theorem 1 (Subject reduction for closures [16]).
Let $\eta :' \Delta$. If $\eta \vdash F \hookrightarrow W$ and $\Delta \vdash F : \tau$, then $W : \tau$.

3 Hybrid Infrastructure

We briefly recall that the theory Hybrid [1] provides support for a deep embedding of higher-order abstract syntax within Isabelle HOL. In particular, it provides a model of the untyped λ-calculus with constants. Let *con* denote a suitable type of constants. The model comprises a type *expr* together with functions

$$\begin{array}{cc} \mathsf{CON} :: con \Rightarrow expr & \$\$ :: expr \Rightarrow expr \Rightarrow expr \\ \mathsf{VAR} :: nat \Rightarrow expr & \mathsf{lambda} :: (expr \Rightarrow expr) \Rightarrow expr \end{array}$$

and two predicates $\mathsf{proper} :: expr \Rightarrow bool$ and $\mathsf{abstr} :: (expr \Rightarrow expr) \Rightarrow bool$. The elements of *expr* which satisfy proper are in one-to-one correspondence with the terms of the untyped λ-calculus modulo α-equivalence. The function CON is the inclusion of constants into terms, VAR is the enumeration of an infinite supply of free variables, and \$\$ is application. The function lambda is declared as a binder and we write $\mathsf{LAM}\, v.\, e$ for $\mathsf{lambda}\ (\lambda v.\, e)$.

For this data to faithfully represent the syntax of the untyped λ-calculus, it must be that CON, VAR, \$\$ are injective on proper expressions and furthermore, lambda is injective on some suitable subset of $expr \Rightarrow expr$. This cannot the whole of $expr \Rightarrow expr$ for cardinality reasons. In fact, we need only a small fragment of the set. The predicate abstr identifies those functions which are sufficiently parametric to be realized as the body of a λ-term, and lambda is injective on these. This predicate can be seen as a full HOAS counterpart of the valid_1 judgment in [5], but it must be defined at the de Bruijn level, since a higher-order definition would require a theory of n-ary abstractions, which is the object of current research.

There is a strong tradition in the HOL community of making extensions *by definition* wherever possible. This ensures the consistency of the logic relative to a small axiomatic core. Hybrid is implemented in a definitional style using a translation into de Bruijn notation. The type *expr* is defined by the grammar

$$expr ::= \mathsf{CON}\ con \mid \mathsf{VAR}\ var \mid \mathsf{BND}\ bnd \mid expr\ \$\$\ expr \mid \mathsf{ABS}\ expr$$

The translation of terms is best explained by example. Let $T_O = \Lambda V_1.\, \Lambda V_2.\, V_1\ V_3$ be an expression in the concrete syntax of the λ-calculus. This is rendered in Hybrid as $T_H = \mathsf{LAM}\, v_1.\, (\mathsf{LAM}\, v_2.\, (v_1\ \$\$\ \mathsf{VAR}\ 3))$ — note the difference between the treatment of free variables and of bound variables. Expanding the binder,

this expression is by definition $\mathsf{lambda}\ (\lambda\, v_1.\, (\mathsf{lambda}\ \lambda\, v_2.\, (v_1\ \$\$\ \mathsf{VAR}\ 3)))$, where λv_i is meta-abstraction. The function $\mathsf{lambda} :: (expr \Rightarrow expr) \Rightarrow expr$ is defined so as to map any function satisfying abstr to a corresponding proper de Bruijn expression. Again, it is defined as an inductive relation on the underlying representation and then proven to be functional. The expression T_H is reduced by higher-order rewriting to the de Bruijn term $\mathsf{ABS}\ (\mathsf{ABS}\ (\mathsf{BND}\ 1\ \$\$\ \mathsf{VAR}\ 3))$. Given these definitions, the essential properties of Hybrid expressions can be proved as theorems from the properties of the underlying de Bruijn representation: for instance, the injectivity of lambda

$$[\![\ \mathsf{abstr}\ E;\ \mathsf{abstr}\ F\ ]\!] \Longrightarrow (\mathsf{LAM}\, v.\, E\ v = \mathsf{LAM}\, v.\, F\ v) = (E = F) \qquad INJ$$

and extensionality $[\![\ \mathsf{abstr}\ E;\ \mathsf{abstr}\ F;\ \forall i.\, E\ (\mathsf{VAR}\ i) = F\ (\mathsf{VAR}\ i)\ ]\!] \Longrightarrow E = F$. Several principles of induction over proper expressions are also provable.

3.1 Coding the Syntax of OL System in Hybrid

We begin by showing how to represent (a fragment of) Mini-ML in Hybrid. In order to render the syntax of the source language in HOAS format we need constants for abstraction, application and fix point, say $cAPP$, $cABS$ and $cFIX$. Recall that in the meta-language application is denoted by infix \$\$, and abstraction by LAM. Then the source language corresponds to the grammar:

$$e ::= v \mid cABS\ \$\$\ (\mathsf{LAM}\, v.\, e\, v) \mid cAPP\ \$\$\ e_1\ \$\$\ e_2 \mid cFIX\ \$\$\ (\mathsf{LAM}\, v.\, e\, v)$$

Thus, we declare these constants to belong to *con* and then make the following *definitions*:

$$\begin{array}{cc} @\ ::\ [\, exp, exp\,] \Rightarrow exp & \mathsf{lam}\ ::\ (exp \Rightarrow exp) \Rightarrow exp \\ e_1\ @\ e_2 == \mathsf{CON}\ cAPP\ \$\$\ e_1\ \$\$\ e_2 & \mathbf{lam}\ x\,.\ E\ x == \mathsf{CON}\ cABS\ \$\$\ \mathsf{LAM}\, x.\, e\ E \end{array}$$

and similarly for the fix point, where lam (resp. fix) is indeed an Isabelle HOL binder. For example, the "real" underlying form of $\mathbf{fix}\ x\,.\ \mathbf{lam}\ y\,.\ x\ @\ y$ is

$$\mathsf{CON}\ cFIX\ \$\$\ (\mathsf{LAM}\, x.\, \mathsf{CON}\ cABS\ \$\$\ \mathsf{LAM}\, y.\, (\mathsf{CON}\ cAPP\ \$\$\ x\ \$\$\ y)$$

It is now possible to *prove* the freeness properties of constructors.

MC-Theorem 1. *The constructors have distinct images; for example,* $\mathbf{lam}\ x\,.\ E\ x \neq \mathbf{fix}\ x\,.\ E'\ x$. *Furthermore, every* binding *constructor is injective on abstractions; for example,* $[\![\ \mathsf{abstr}\ E;\ \mathsf{abstr}\ E'\ ]\!] \Longrightarrow (\mathbf{fix}\ x\,.\ E\ x = \mathbf{fix}\ x\,.\ E'\ x) = (E = E')$.

This is proven via Isabelle HOL's simplification, using property INJ.

Although dB expressions are strictly first-order, we still need to encode them as Hybrid expressions. In fact, we will use for compilation a judgment which is parametric in (higher-order) terms, dB expression and values. Therefore they all have to be interpreted via the SL universal quantification and consequently need

to be synonymous to Hybrid (proper) terms, to make the universal quantification consistent[2]. The encoding is trivial and the details omitted.

The informal definition of environments and closure is by mutual recursion. Since our aim here is also to show how Hybrid expressions can coexist with regular Isabelle HOL ones, we will use Isabelle HOL mechanism for mutually inductive datatypes. This brings about the declaration of a datatype of polymorphic environments, intended to be instantiated with a value. Environments can be now mutually recursive with closures, where the type synonymous exp' is retained for forward compatibility with Subsection 5.1:

$$\begin{array}{rl} \mathit{datatype}\ (\alpha\ \mathit{env}) & ::= \cdot \mid (\alpha\ \mathit{env})\ ;\ \alpha \mid (\alpha\ \mathit{env}) + \mathit{exp}' \\ \mathit{and} \quad \alpha\ \mathit{clos} & ::= \mathsf{mk\_clo}\ (\alpha\ \mathit{env})\ \mathit{exp}' \end{array}$$

Then we introduce in *con* a constructor, say $cCLO(val\ clos)$, which encapsulates a Isabelle HOL closure. Finally, we can *define* a Hybrid closure as a constant of type $[(val\ env), exp'] \Rightarrow val$, defined as $\{\eta, F\} == \mathsf{CON}\ (cCLO\ (\mathsf{mk\_clo}\ \eta\ F))$. Thanks to this rather cumbersome encoding, we can establish freeness properties of closures as in Theorem 1.

4 Multi-level Architecture

In previous work [1, 17], we chose to work in a single level, implementing every judgment as a (co)inductive definition in Isabelle HOL, but exploiting the form of HOAS that our package supports. While the tool seemed successful in dealing with predicates over *closed* terms, say evaluation, we had to resort to a more traditional encoding, i.e. via explicit environments, with respect to judgments involving open ones such as typing. As we have mentioned earlier, a two-level approach solves this dilemma.

4.1 Encoding the Specification Logic

We introduce our specification logic, namely a fragment of second-order hereditary Harrop formulae, which is sufficient for the encoding of our case-study.

$$\begin{array}{rl} \mathit{Clauses}\ D & ::= A \mid D_1\ \mathsf{and}\ D_2 \mid G\ \mathsf{imp}\ A \mid \mathsf{tt} \mid \mathsf{all}\, x.\, D \\ \mathit{Goals}\ G & ::= A \mid G_1\ \mathsf{and}\ G_2 \mid A\ \mathsf{imp}\ G \mid \mathsf{tt} \mid \mathsf{all}\, x.\, G \end{array}$$

This syntax translates immediately in a Isabelle HOL datatype:

$$\mathit{datatype}\ oo ::= \mathsf{tt} \mid \langle \mathit{atm} \rangle \mid oo \quad \mathsf{and} \quad oo \mid \mathit{atm} \quad \mathsf{imp} \quad oo \mid \mathit{all}\ (\mathsf{prpr} \Rightarrow oo)$$

where $\langle \_ \rangle$ coerces atoms into propositions. The universal quantifier is intended to range over all proper Hybrid terms. In analogy with logic programming, it will be left implicit in clauses.

[2] This is because Hybrid, in its current formulation, provides only what amounts to an *untyped* meta-language. This is being addressed in the next version

This logic is so simple that its proof-system can be modeled with a logic programmer interpreter; in fact, for such a logic, *uniform provability* [15] (of a sequent-calculus) is complete. We give the following definition of provability:

$$\begin{aligned}
\mathit{Inductive}\ _ \rhd_{_}\ _ \ ::\ & [\,\mathit{nat}, (\mathit{atmlist}), \mathit{oo}\,] \Rightarrow \mathit{bool} \\
& \Longrightarrow \Phi \rhd_n \mathsf{tt} \\
[\![\, \Phi \rhd_n G_1; \Phi \rhd_n G_2 \,]\!] & \Longrightarrow \Phi \rhd_{n+1} (G_1 \ \mathsf{and}\ G_2) \\
[\![\, \forall x. \Phi \rhd_n (G\ x) \,]\!] & \Longrightarrow \Phi \rhd_{n+1} (\mathsf{all}\, x.\, G\ x) \\
[\![\, \Phi, A \rhd_n G \,]\!] & \Longrightarrow \Phi \rhd_{n+1} (A \ \mathsf{imp}\ G) \\
[\![\, A \in \Phi \,]\!] & \Longrightarrow \Phi \rhd_n \langle A \rangle \\
[\![\, A \longleftarrow G; \Phi \rhd_n G \,]\!] & \Longrightarrow \Phi \rhd_{n+1} \langle A \rangle
\end{aligned}$$

Note the following:

- Only atomic antecedent are required in implications which therefore yield only atomic contexts.
- Atoms are provable either by assumptions or via *backchaining* over a set of Prolog-like rules, which encode the properties of the object-logic in question. The suggestive notation $A \longleftarrow G$ corresponds to an inductive definition of a set `prog` of type $[\,\mathit{atm}, \mathit{oo}\,] \Rightarrow \mathit{bool}$, see Subsection 4.2. The sequent calculus is parametric in those clauses and so are its meta-theoretical properties.
- Sequents are decorated with natural numbers which represent the *height* of a proof; this measure allows reasoning by complete induction.
- For convenience we define $\Phi \rhd G$ iff $\exists n.\ \Phi \rhd_n G$ and $\rhd G$ iff $\cdot \rhd G$.
- The very fact that provability is inductive makes available *inversion principles* as elimination rules of the aforementioned definition. In Isabelle HOL (as well as in Coq) case analysis is particularly well-supported as part of the datatype/inductive package. For example the inversion theorem that analyses the shape of a derivation ending in an atom from the empty context is obtained simply with a call to the built-in `mk_cases` function, which specializes the elimination principle to the given constructors:

$$[\![\, \cdot \rhd_i \langle A \rangle; \bigwedge G\ j. [\![\, A \longleftarrow G; \cdot \rhd_j G; i = j + 1 \,]\!] \Longrightarrow P \,]\!] \Longrightarrow P$$

- The adequacy of the encoding of the SL can be established as in [14].

MC-Theorem 2 (Structural Rules). *The following rules are provable:*

1. *Weakening for numerical bounds:* $[\![\, \Phi \rhd_n G;\ n < m \,]\!] \Longrightarrow \Phi \rhd_m G$
2. *Context weakening:* $[\![\, \Phi \rhd G;\ \Phi \subseteq \Phi' \,]\!] \Longrightarrow \Phi' \rhd G$
3. *(Atomic) cut:* $[\![\, \Phi, A \rhd G;\ \Phi \rhd \langle A \rangle \,]\!] \Longrightarrow \Phi \rhd G$

Proof.

1. The proof, by structural induction on sequents, consists of a one-line call to an automatic tactic using the elimination rule for successor (from the Isabelle HOL library) and the introduction rules for the sequent calculus. This compared to the much longer proof of the same statement in Coq reported in [6].

2. By a similar fully automated induction on the structure of the sequent derivation.
3. Cut is a corollary of the following lemma:

$$[\![\ \Phi' \rhd_i G;\ \Phi' = set\ (\Phi, A);\ \Phi \rhd_j \langle A \rangle\ ]\!] \Longrightarrow \Phi \rhd_{i+j} G$$

easily proven by induction on the structure of the derivation of $\Phi' \rhd_i G$, using library facts relating set and list memberships.

4.2 Encoding the Object Logic

We introduce a datatype `atm` to encode the atomic formulae of the OL, which in this case study includes

$$datatype\ atm ::= exp : tp \mid exp \Downarrow exp \mid (val\ env) \vdash exp' \leftrightarrow exp \mid val \Leftrightarrow exp$$

We can now give the clauses for the OL deductive systems; we start with typing and evaluation:

$$\begin{aligned}
&Inductive\ _ \longleftarrow _ :: [\, atm, oo \,] \Rightarrow bool \\
\Longrightarrow\ & (E_1\ @\ E_2) : T \longleftarrow \langle E_1 : (T'\ \mathsf{arrow}\ T)\rangle\ \mathsf{and}\ \langle E_2 : T'\rangle \\
[\![\ \mathsf{abstr}\ E\ ]\!] \Longrightarrow\ & (\mathbf{lam}\ x\,.\ E\ x) : (T_1\ \mathsf{arrow}\ T_2) \longleftarrow \mathsf{all}\, x.\, (x : T_1)\ \mathsf{imp}\ \langle (E\ x) : T_2\rangle \\
[\![\ \mathsf{abstr}\ E\ ]\!] \Longrightarrow\ & (\mathbf{fix}\ x\,.\ E\ x) : T \longleftarrow \mathsf{all}\, x.\, (x : T)\ \mathsf{imp}\ \langle (E\ x) : T\rangle
\end{aligned}$$

$$\begin{aligned}
[\![\ \mathsf{abstr}\ E\ ]\!] \Longrightarrow\ & \mathbf{lam}\ x\,.\ E\ x \Downarrow \mathbf{lam}\ x\,.\ E\ x \longleftarrow \mathsf{tt} \\
[\![\ \mathsf{abstr}\ E_1'\ ]\!] \Longrightarrow\ & E_1\ @\ E_1 \Downarrow V \longleftarrow \\
& \langle E_1 \Downarrow \mathbf{lam}\ x\,.\ E_1'\ x\rangle\ \mathsf{and}\ \langle E_2 \Downarrow V_2\rangle\ \mathsf{and}\ \langle (E_1'\ V_2) \Downarrow V\rangle \\
[\![\ \mathsf{abstr}\ E\ ]\!] \Longrightarrow\ & \mathbf{fix}\ x\,.\ E\ x \Downarrow V \longleftarrow \langle E\ (\mathbf{fix}\ x\,.\ E\ x) \Downarrow V\rangle
\end{aligned}$$

Note the presence of the *abstraction* annotations as Isabelle HOL premises in rules mentioning binding construct. This in turn allows to simulate definitional reflection via the built-in elimination rules of the `prog` inductive definition *without* the use of additional axioms. For example inversion on the function typing rule is:

$$[\![\ \mathbf{lam}\ x\,.\ E\ x : \tau \longleftarrow G; \bigwedge F, T_1, T_2.\ [\![\ \mathsf{abstr}\ F; G = \mathsf{all}\, x.\, (x : T_1)\ \mathsf{imp}\, \langle F\ x : T_2\rangle);\ \mathsf{lambda}\ E = \mathsf{lambda}\ F; \tau = (T_1 \rightarrow T_2)\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$$

Note also how the inversion principle has an explicit equation $\mathsf{lambda}\ E = \mathsf{lambda}\ F$ (whereas definitional reflection employs unification) and those are solvable under the assumption that the body of a lambda term is well-behaved, i.e. an abstraction.

Now we can address the meta-theory, starting, for example, with the subject reduction theorem:

MC-Theorem 3 (OL Subject Reduction).
$\forall n.\, (\cdot \rhd_n E \Downarrow V) \Longrightarrow \forall T.\, (\rhd E : T) \rightarrow (\rhd V : T)$.

Proof. The proof is by complete induction on the height of the derivation of evaluation, analogously to [6] (except with an appeal to Theorem 1 rather than to the distinctness axioms).

As we remarked, the main reason to reference provability is the intrinsic incompatibility of induction with hypothetical (non-stratifiable) judgments. Since the definition of evaluation makes no use of hypothetical judgments, it is perfectly acceptable at the meta-level, that is, we can directly give an inductive definition for it.

$$\begin{array}{rl} & \mathit{Inductive}\ \_ \Downarrow \_ \ ::\ [\,\mathit{exp}, \mathit{exp}\,] \Rightarrow \mathit{bool} \\ [\![\ E_1 \Downarrow \mathbf{lam}\ x\,.\ E'\ x;\ \mathsf{abstr}\ E'; & \\ \quad E_2 \Downarrow V_2;\ (E'\ V_2) \Downarrow V\ ]\!] & \Longrightarrow (E_1\ @\ E_2) \Downarrow V \\ [\![\ \mathsf{abstr}\ E\ ]\!] & \Longrightarrow \mathbf{lam}\ x\,.\ E\ x \Downarrow \mathbf{lam}\ x\,.\ E\ x \\ [\![\ \mathsf{abstr}\ E;\ (E\ (\mathbf{fix}\ x\,.\ E)) \Downarrow V\ ]\!] & \Longrightarrow \mathbf{fix}\ x\,.\ E \Downarrow V \end{array}$$

Moreover, it is easy to show (formally) the two encodings equivalent:

MC-Theorem 4. $E \Downarrow V$ *iff* $\cdot \rhd_n E \Downarrow V$.

Proof. Left-to-right holds by structural induction. The converse is by complete induction.

This suggest we delegate to the OL level *only* those judgments, such as typing, which would not be inductive at the meta-level. This has the benefit of limiting the indirect-ness of using an explicit SL. Moreover, it has the further advantage of replacing complete with structural induction, which is better behaved from a proof-search point of view. Complete induction, in fact, places an additional burden to the user by requiring him/her to provide the correct instantiation for the height of the derivation in question. Thus subject reduction at the meta-level has the form:

MC-Theorem 5 (Meta-level Subject Reduction).
$E \Downarrow V \Longrightarrow \forall T.\, (\rhd\ E : T) \rightarrow (\rhd\ V : T)$.

Proof. By structural induction on evaluation, using only inversion principles on provability and OL typing.

5 Formal Verification of Compilation

Similarly, we implement evaluation to closures $\eta \vdash F \hookrightarrow W$ at the meta-level as a straightforward (i.e. entirely first-order) inductive definition, whose rules we omit. Compilation to dB expressions and values is instead represented at the OL level.

$$\begin{array}{rl} & \Longrightarrow \eta \vdash (F_1\ @'\ F_2) \leftrightarrow (E_1\ @\ E_2) \longleftarrow \\ & \qquad \langle \eta \vdash F_1 \leftrightarrow E_1 \rangle\ \mathsf{and}\ \langle \eta \vdash F_2 \leftrightarrow E_2 \rangle \\ [\![\ \mathsf{abstr}\ E\ ]\!] & \Longrightarrow \eta \vdash \mathbf{lam}'\ F \leftrightarrow (\mathbf{lam}\ x\,.\ E\ x) \longleftarrow \\ & \qquad \mathsf{all}\, w.\, \mathsf{all}\, x.\, w \Leftrightarrow x\ \mathsf{imp}\ \langle (\eta; w) \vdash F \leftrightarrow (E\ x) \rangle \end{array}$$

$$\begin{aligned}
[\![\ \mathsf{abstr}\ E\ ]\!] &\Longrightarrow \eta \vdash \mathbf{fix}'\ F \leftrightarrow (\mathbf{fix}\ x\,.\ E\ x) \longleftarrow \\
&\qquad\qquad \mathsf{all}\ f.\, \mathsf{all}\ x.\, \eta \vdash f \leftrightarrow x\ \mathsf{imp}\ \langle (\eta + f) \vdash F \leftrightarrow (E\ x)\rangle \\
&\Longrightarrow (\eta; W) \vdash 1 \leftrightarrow E \longleftarrow \langle W \Leftrightarrow E\rangle \\
&\Longrightarrow (\eta; W) \vdash F \uparrow\!\leftrightarrow E \longleftarrow \langle \eta \vdash F \leftrightarrow E\rangle \\
[\![\ \mathsf{abstr}\ E\ ]\!] &\Longrightarrow \{\eta, \mathbf{lam}'\ F\} \Leftrightarrow \mathbf{lam}\ x\,.\ E\ x \longleftarrow \langle \eta \vdash \mathbf{lam}'\ F \leftrightarrow \mathbf{lam}\ x\,.\ E\ x\rangle
\end{aligned}$$

We can now tackle the verification of compiler correctness:

MC-Theorem 6 (Soundness of Compilation).
$E \Downarrow V \Longrightarrow \forall n\ \eta\ F.(\cdot \rhd_n \eta \vdash F \leftrightarrow E) \rightarrow \exists W.\eta \vdash F \hookrightarrow W \wedge (\rhd\ W \Leftrightarrow V)$

Proof. The informal proof proceeds by lexicographic induction on the pair consisting of the derivation $\mathcal{D}$ of $E \Downarrow V$ and $\mathcal{C}$ of $\eta \vdash F \leftrightarrow E$. We examine one case, both formally and informally to demonstrate the use of hypothetical judgments and its realization in the system. Let $\mathcal{D}$ end in **ev_app** and assume $\eta \vdash F \leftrightarrow e_1\ @\ e_2$. Inversion on the latter yields three cases. Let's consider only the last one, where $F = F_1\ @'\ F_2$ such that $\eta \vdash F_1 \leftrightarrow e_1$ and $\eta \vdash F_2 \leftrightarrow e_2$. By IH, $\eta \vdash F_2 \hookrightarrow W_2$ and $W_2 \Leftrightarrow v_2$. Moreover, $\eta \vdash F_1 \hookrightarrow W_1$ and $W_1 \Leftrightarrow \mathbf{lam}\ x\,.\ e_1'$. By inversion, $W_1 = \{\eta_1, \mathbf{lam}'\ F_1'\}$ and $\eta_1 \vdash \mathbf{lam}'\ F_1' \leftrightarrow \mathbf{lam}\ x\,.\ e_1'$. By a further inversion we have a proof of $\eta_1; w \vdash F_1' \leftrightarrow e_1'$ under the parametric assumption $w \Leftrightarrow x$. Now substitute W_2 for w and v_2 for x: since $W_2 \Leftrightarrow v_2$ holds, we infer $\eta_1; W_2 \vdash F_1' \leftrightarrow [v_2/x]e_1'$. We can now apply the IH once more yielding $\eta_1; W_2 \vdash F_1' \hookrightarrow W_3$ and $W_3 \Leftrightarrow v$. The case is finished with an appeal to rule **feval_app**.

In the formal development we proceed by a nested induction, the outermost on the structure of $E \Downarrow V$ and the innermost by complete induction on n, representing the height of the SL's proof of $\eta \vdash F \leftrightarrow E$. Let $IH(E,V) = \forall n\ \eta\ F.\cdot \rhd_n \langle \eta \vdash F \leftrightarrow E\rangle \rightarrow \exists W.\eta \vdash F \hookrightarrow W \wedge \rhd\ \langle W \Leftrightarrow V\rangle$ and $G = \exists W.\eta \vdash F_1\ @'\ F_2 \hookrightarrow W \wedge \rhd\ \langle W \Leftrightarrow V\rangle$. Then the case for application from the outermost induction is, omitting the innermost IH which is not used here:

$$[\![\ E_1 \Downarrow \mathbf{lam}\ x\,.\ E'\ x;\ IH(E_1, \mathbf{lam}\ x\,.\ E_1'\ x);\ \mathsf{abstr}\ E';\ E_2 \Downarrow V_2;\ IH(E_2, V_2);$$
$$(E'\ V_2) \Downarrow V;\ IH((E'\ V_2), V);\ \cdot \rhd_n \langle \eta \vdash F_1 \leftrightarrow E_1\rangle\ \mathsf{and}\ \langle \eta \vdash F_2 \leftrightarrow E_2\rangle\ ]\!] \Longrightarrow G$$

Eliminating the SL conjunction "and" and applying twice the IH, we get:

$$[\![\ \ldots \exists W.\eta \vdash F_1 \hookrightarrow W \wedge \rhd\ \langle W \Leftrightarrow \mathbf{lam}\ x\,.\ E'\ x\rangle;\ \exists W.\eta \vdash F_2 \hookrightarrow W \wedge \ldots\ ]\!] \Longrightarrow G$$

Now we perform inversion on $W \Leftrightarrow \mathbf{lam}\ x\,.\ E'\ x$ and simplification:

$$[\![\ \ldots \rhd\ \langle \eta \vdash \mathbf{lam}'\ F_1' \leftrightarrow \mathbf{lam}\ x\,.\ E_1'\ x\rangle \ldots\ ]\!] \Longrightarrow G$$

More inversion on $\eta_1 \vdash \mathbf{lam}'\ F_1' \leftrightarrow \mathbf{lam}\ x\,.\ E_1'\ x$, plus Hybrid injectivity to solve $\mathsf{lambda}\ E = \mathsf{lambda}\ E_1'$:

$$\begin{aligned}
[\![\ \ldots \eta \vdash F_1 \hookrightarrow \{\eta_1, \mathbf{lam}'\ F_1'\};\ \eta \vdash F_2 \hookrightarrow W_2;\ \rhd\ \langle W_2 \Leftrightarrow V_2\rangle; \\
\rhd\ \mathsf{all}\ w.\, \mathsf{all}\ x.\, (w \Leftrightarrow x\ \mathsf{imp}\ \langle \eta_1;\ w \vdash F_1' \leftrightarrow (E_1'\ x)\rangle)\ ]\!] \Longrightarrow G
\end{aligned}$$

We now invert on the (proof of the) parametric and hypothetical judgment and then instantiate w with W_2 and x with V_2:

$$[\![\ldots \eta \vdash F_1 \hookrightarrow \{\eta_1, \mathbf{lam}'\ F_1'\};\ \eta \vdash F_2 \hookrightarrow W_2;\ \rhd \langle W_2 \Leftrightarrow V_2 \rangle; \\ W_2 \Leftrightarrow V_2 \rhd \langle \eta_1; W_2 \vdash F_1' \leftrightarrow (E_1'\ V_2) \rangle\ ]\!] \Longrightarrow G$$

After a cut we can apply the IH to $\eta_1; W_2 \vdash F_1' \leftrightarrow (E_1'\ V_2)$ yielding $\eta_1; W_2 \vdash F_1' \hookrightarrow W$. Then the proof is concluded with the introduction rule for application.

The converse does not contribute any new ideas and we leave it to the on-line documentation:

MC-Theorem 7 (Completeness of Compilation).
$\eta \vdash F \hookrightarrow W \Longrightarrow \forall n\ E.\, (\cdot \rhd_n \eta \vdash F \leftrightarrow E) \rightarrow \exists V.\, E \Downarrow V \wedge \rhd W \Leftrightarrow V$

Proof. By structural induction on $\eta \vdash F \hookrightarrow W$ and inversion on $\cdot \rhd_n \eta \vdash F \leftrightarrow E$.

5.1 Implementation of Subject Reduction for Closures

We now turn to the co-inductive part. Ideally, we would implement closures and environment as a *co-datatype*; indeed, this is possible in Isabelle/ZF, but not at the moment in Isabelle HOL. We then resort to a definitional approach where we introduce a Hybrid constant $\{\_, \_\}^\infty :: [\,(val\ env), exp'\,] \Rightarrow val$ and we prove it to be injective as usual. There are alternatives; for example we have implemented on operational semantics with *recursive* closures, but we present here the above to be faithful to [16]. On the other hand, as remarked in Subsection 2.2, since we will not need the + environment constructor anymore, nothing prevents us from encoding here dB expressions with a Isabelle HOL datatype:

$$datatype\ exp' ::= 1 \mid exp' \uparrow \mid \mathbf{lam}'\ exp' \mid exp'\ @'\ exp' \mid \mathbf{fix}'\ exp'$$

We only mention the new "circular" rule:

$$Inductive\ \_ \vdash \_ \hookrightarrow \_\ ::\ [\,(val\ env), exp', val\,] \Rightarrow bool \\ [\![\ cl = \{(\eta; cl), \mathbf{lam}'\ F\}^\infty\ ]\!] \Longrightarrow \eta \vdash \mathbf{fix}'\ (\mathbf{lam}'\ F) \hookrightarrow cl$$

We declare a standard HOL datatype for types environments *tenv* and we encode the judgment $\Delta \vdash F : \tau$ with an inductive definitions of type $[\![\ tenv, exp', tp\ ]\!] \Rightarrow bool$, whose rules are obvious and again omitted. More interestingly, we introduce typing of closures and type consistency of value and type environments as a *mutually co-inductive* definition:

$$Coinductive\ \_ : \_\ ::\ [\,val, tp\,] \Rightarrow bool \\ \_ :' \_\ ::\ [\,(val\ env), tenv\,] \Rightarrow bool \\ [\![\ \Delta \vdash \mathbf{lam}'\ F : \tau;\ \eta :' \Delta\ ]\!] \Longrightarrow \{\eta, \mathbf{lam}'\ F\}^\infty : \tau \\ \Longrightarrow \cdot :' \cdot \\ [\![\ W : \tau;\ \eta :' \Delta\ ]\!] \Longrightarrow (\eta; W) :' (\Delta; \tau)$$

MC-Theorem 8. $\eta \vdash F \hookrightarrow W \Longrightarrow \forall T.\, \eta :' \Delta \rightarrow (\Delta \vdash F : \tau \rightarrow W : \tau)$

Proof. By structural induction on evaluation, where each case is proven with an appeal to an automatic tactic, which uses appropriate elimination rules. The only delicate case is the fix point, where we need to prove:

$$[\![\ cl = \{(\eta; cl), \mathbf{lam}'\ F\}^{\infty};\ \eta :' \Delta;\ \Delta \vdash \mathbf{fix}'\ (\mathbf{lam}'\ F) : \tau\ ]\!] \Longrightarrow cl : \tau$$

In Isabelle HOL (co)induction is realized set theoretically via the Knaster-Tarski's construction and the user provides the right set to be checked for *density* w.r.t. the rule set. Since our definition is by mutual co-induction, the greatest fix point is constructed as a disjoint sum. Thus, the right set turns out to be $\{\mathsf{Inr}(\eta; cl), \mathsf{Inl}(\Delta; \tau)\}$.

6 Related Work

We have so far concentrated on $FO\lambda^{\Delta IN}$, but the other major contender in the field is the *Twelf* project [20]. Meta-reasoning can be carried over in two complementary styles. In an interactive one [19], LF is used to specify a meta-theorem as a relation between judgments, while a logic programming-like interpretation provides the operational. Finally, external checks (termination, moded-ness, totality) verify that the given relation is indeed a realizer for that theorem. The second approach [21] is built on the idea of devising an explicit meta-meta-logic for reasoning (inductively) about logical frameworks, in a fully automated way. $\mathcal{M}_\omega$ is a constructive first-order logic, whose quantifiers range over possibly open LF object over a signature. By the adequacy of the encoding, the proof of the existence of the appropriate LF object(s) guarantees the proof of the corresponding object-level property. It must be remarked that *Twelf* is not programmable by tactics, nor does it support co-induction.

Other architectures are essentially one level. For lack of space, we refer to the review in [1], but we just mention Honsell et al.'s framework [12], which embraces an *axiomatic* approach to meta-reasoning with HOAS. It consists of higher-order logic extended with a set of axioms parametric to a HOAS signature, including the reification of key properties of names akin to *freshness*. A limited form of recursion over HOAS syntax is also assumed. Similarly the FM approach [7] aims to be a foundation of programming and reasoning with *names* in a one-level architecture. It would be interesting to look at using a version of the "New" quantifier in the specification logic, especially for those applications where the behavior of the object-logic binder is not faithfully mirrored by a traditional universal quantification at the SL-level, for example the π-calculus.

7 Conclusions and Future Work

We have presented a multi-level architecture to allow (co)inductive reasoning about objects defined via HOAS in a well-known environment such as Isabelle HOL. Similarly to [6] this has several benefits:

- It is possible to replicate in an well-understood and interactive setting the style of proof of $FO\lambda^{\Delta\mathbb{N}}$, so all results are proven without "technical" lemmas foreign to the mathematics of the problem.
- Results about the specification logic, such as cut elimination, are proven once and for all, if we are happy with that logic. Otherwise, different logics (say linear) can be employed, without changing infrastructure. This would allow, for example, the utilization of the most elegant encodings of the meta-theory of functional programming with references proposed, for instance, in [14].

Differently to [6], our architecture is based not directly on a standard proof-assistant, but on a package which builds a HOAS meta-language on top of such a system. This allows us not to rely on any axiomatic assumptions, such as freeness of HOAS constructors and extensionality properties at higher types, which are now theorems. Another difference is the mixing of meta-level and OL specifications, which we have shown makes proofs more easily mechanizable and allows us to use co-induction which is still unaccounted for in $FO\lambda^{\Delta\mathbb{N}}$. Finally, by the simple reason that the Hybrid system sits on top of Isabelle HOL, we benefit of the higher degree of automation of the latter.

As far as future work is concerned, we plan to further pursue our case study by finally compiling our target language into a CAM-like abstract machine, as in [10]. We shall also investigate co-inductive issues in compilation, starting with verifying the equivalence between the standard operational semantics and the one with non-well founded closures.

Note that in this case study we only needed to induct — either directly, or on the height of derivations — over *closed* terms, although we extensively reasoned in presence of hypothetical judgments. Inducting HOAS-style over open terms is a major challenge [21]; in this setting *generic* judgments are particularly problematic, but can be dealt with by switching to a more expressive SL, based on a eigenvariable encoding [14]. While it is already simple enough to implement such a logic, the new theory of *n-ary abstractions* which underlines the next version of the Hybrid infrastructure will directly support this syntax, as well as a form of Isabelle HOL-like datatypes over HOAS signatures. With that in place, we will be able, for example, to revisit in a full HOAS style the material in [17].

Source files for the Isabelle HOL code can be found at

`www.mcs.le.ac.uk/~amomigliano/isabelle/2Levels/Compile/main.html`

Acknowledgments This paper has benefited from referees comments and discussions with Roy Crole, Dale Miller, Frank Pfenning, Carsten Schürmann and Amy Felty, who kindly made available to us the Coq proof script of [6].

References

[1] S. Ambler, R. Crole, and A. Momigliano. Combining higher order abstract syntax with tactical theorem proving and (co)induction. In V. A. Carreño, editor, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*, volume 2342 of *LNCS*. Springer Verlag, 2002.

[2] S. Boutin. Proving correctness of the translation from mini-ML to the CAM with the Coq proof development system. Technical Report RR-2536, Inria, Institut National de Recherche en Informatique et en Automatique, 1995.
[3] B. Ciesielski and M. Wand. Using the theorem prover Isabelle-91 to verify a simple proof of compiler correctness. Technical Report NU-CCS-91-20, College of Computer Science, Northeastern University, Dec. 1991.
[4] J. Despeyroux. Proof of translation in natural semantics. In *Proceedings of LICS'86*, pages 193–205, Cambridge, MA, 1986. IEEE Computer Society Press.
[5] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 124–138, Edinburgh, Scotland, Apr. 1995. Springer-Verlag LNCS 902.
[6] A. Felty. Two-level meta-reasoning in Coq. In V. A. Carreño, editor, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*, volume 2342 of *LNCS*. Springer Verlag, 2002.
[7] M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, 1999. IEEE Computer Society Press.
[8] L. Hallnas. Partial inductive definitions. *TCS*, 87(1):115–147, July 1991.
[9] J. Hannan and D. Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.
[10] J. Hannan and F. Pfenning. Compiler verification in LF. In A. Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992.
[11] P. H. Hartel and L. Moreau. Formalizing the safety of Java,the Java Virtual Machine, and Java Card. *ACMCS*, 33(4):517–558, Dec. 2001.
[12] F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In *Proc. ICALP'01*, number 2076 in LNCS, pages 963–978. Springer-Verlag, 2001.
[13] D. Lester and S. Mintchev. Towards machine–checked compiler correctness for higher–order pure functional languages. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic*, pages 369–381. Springer-Verlag LNCS 933, 1995.
[14] R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, 2002.
[15] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
[16] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.
[17] A. Momigliano, S. Ambler, and R. Crole. A Hybrid encoding of Howe's method for establishing congruence of bisimilarity. *ENTCS*, 70(2), 2002.
[18] F. Pfenning. *Computation and Deduction*. Cambridge University Press, 2000. In preparation. Draft from April 1997 available electronically.
[19] F. Pfenning and E. Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 537–551. Springer-Verlag LNAI 607.
[20] F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of CADE 16*, pages 202–206. Springer LNAI 1632.
[21] C. Schürmann. *Automating the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie-Mellon University, 2000. CMU-CS-00-146.

Abstraction in Reasoning about Higraph-Based Systems

John Power and Konstantinos Tourlas*

Division of Informatics, The University of Edinburgh, King's Buildings, EH9 3JZ, United Kingdom
{ajp,kxt}@inf.ed.ac.uk

Abstract. Higraphs, a kind of hierarchical graph, underlie a number of sophisticated diagrammatic formalisms, including Statecharts. Naturally arising from hierarchy in higraphs is an abstraction operation known as *zooming out*, which is of profound importance to reasoning about higraph-based systems. We motivate how, in general, the use of zooming in reasoning requires sophisticated extensions to the basic notion of higraph and a careful definition of higraph dynamics (i.e. semantics), which we contribute. Our main results characterise zooming by means of a universal property and establish a precise relationship between the dynamics of a higraph and that of its zoom-out.

1 Introduction

Recent years have witnessed a rapid, ongoing popularisation of diagrammatic notations in the specification, modelling and programming of computing systems. Most notable among them are Statecharts [5], a notation for modelling reactive systems, and the Unified Modelling Language (UML), a family of diagrammatic notations for object-based modelling. Being spatial rather than linear representations of computing systems, diagrams lend themselves to a variety of intuitive structural manipulations, common among which are those implementing filtering and abstraction operations to control the level of detail [10].

Often, such manipulations are employed to assist in the visualisation process [10], as diagrams may grow impractically large even in economic and compact notations. Of particular importance also are uses of filtering and abstraction in the course of reasoning about the represented system. In that case, the user attempts to simplify the reasoning task by considering a higher-level, more abstract diagrammatic representation of the system in question, obtained by discarding detail which is deemed irrelevant to the reasoning argument. Thus, a precise relationship between the form (syntax) and meaning (semantics) of a diagram resulting from filtering or abstraction and those of the original one is pre-requisite to ensuring soundness of reasoning.

* Both authors wish to acknowledge the support of a British Council grant and of AIST Japan. John Power also acknowledges the support of EPSRC grant no. M566333, and Konstantinos Tourlas the support of EPSRC grant no. GR/N12480.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 392–408, 2003.

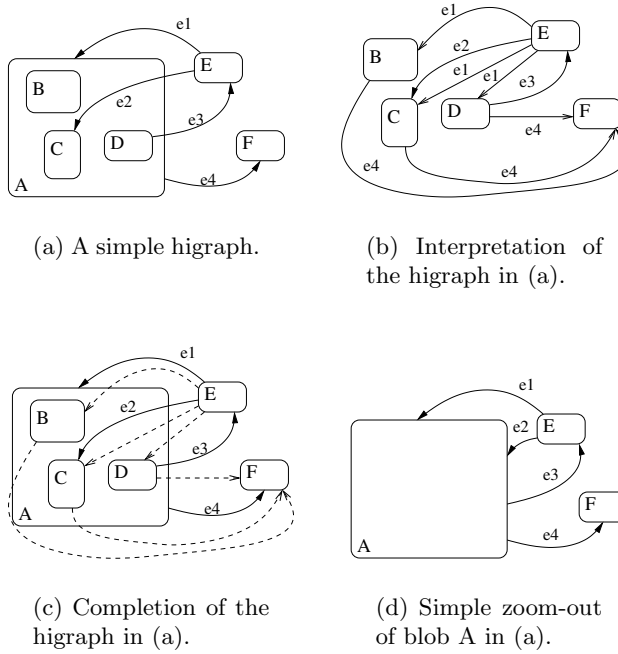
Starting with *higraphs*, which are structures underlying a variety of sophisticated diagrammatic formalisms in computing, we provide such a precise relationship for a practically important filtering operation known as *zooming out*. Doing so, we argue, requires sophisticated extensions to the basic notion of higraph and a careful definition of zooming and of higraph dynamics (i.e. semantics), all of which we contribute. So we give precise definitions of a *meet higraphs*, the dynamics of meet-higraphs, and zooming-out of a meet-higraph. We further contribute two main results. The first characterises one's informal understanding of zooming by means of the universal property of an adjunction. The other establishes how the dynamics of a higraph is reflected in the dynamics of its zoom-out, as required for supporting the use of zooming in reasoning about higraph-based systems.

The work in [4, 1], concerned with compositionality and modularity questions, also addresses some similar semantic issues. It does, however, require that hierarchical graphs are extended to include, for each node, a set of "interface points." Here, we do not wish to rely on this kind of non-standard extension. Instead we develop, explain and formalise our solution in terms only of the devices, known as "stubs" or "loose edges", which are already used in practice and have been adopted in UML and other applications.

Thus, our work relates closely to Harel's original formulation of higraphs [6] and to the notions of "zooming out" and "loose higraph" which were briefly and informally introduced therein. We have formalised these structures in [11, 2, 12]. After recalling our previous analyses and results from [12], our work here extends our theory and develops new results. This is necessary, as we argue below, in order to address important practical applications of higraphs which lie beyond the scope of Harel's original brief treatment. The latter focused zooming on only a limited class of higraphs (those arising from Statecharts), but which are clearly too restricted for other important applications, including the hierarchical entity-relationship (ER) modelling of complex databases, also from [6].

The utility, nonetheless, of the framework of concepts and techniques which we develop here extends beyond higraphs. The concept of *hulls*, for instance, which is introduced and studied below, naturally pertains, more generally than higraphs, to other graph-based notations which feature intersecting vertices. There has been increasing computational interest in such notations, one recent example being the Constraint diagrams [3] for use alongside UML.

In the following section we recall the most basic notions of higraph and zooming, and we motivate the need for developing more subtle higraph structures so as to support uses of zooming in reasoning. *Meet higraphs*, such suitably sophisticated structures, are developed in Section 3. *Hulls*, a related but more widely applicable concept, is independently developed in Section 4. In Section 5, we define a zooming operation appropriate to meet-higraphs and prove the main theorem asserting its intuitive universal property. A notion of dynamics for our higraphs and the main theorem relating the dynamics of a meet higraph and those of its zoom-out is the subject of Section 6. Finally, in the concluding section we remark on how our work can benefit the development of software



(a) A simple higraph.

(b) Interpretation of the higraph in (a).

(c) Completion of the higraph in (a).

(d) Simple zoom-out of blob A in (a).

Fig. 1. Interpretation, completion and zooming on a higraph.

tools for higraph-based, as well as other diagrammatic notations. Owing to space considerations, some proofs and lemmas are delegated to an Appendix.

2 Higraphs, Zooming and Reasoning

Higraphs are structures extending graphs by permitting spatial containment among the nodes. Since their introduction by Harel [6] as a foundation for Statecharts [5], higraphs have rapidly become prominent structures in computing. Beyond Statecharts and UML, their diverse applications include notations for the entity-relationship diagrams [6] of database theory and knowledge representation, and for reasoning with temporal logics [8], as well as programming languages, such Argos [9].

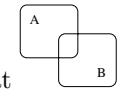
We recall from [11] the definition of an (ordinary) higraph:

Definition 1. *A higraph is a quadruple* $(B, E; s, t : E \to B)$*, where* B *and* E *are posets and* s, t *are monotone functions.* □

We refer to the elements of B as *blobs* and to those of E as *edges*, while the functions provide for each $e \in E$ its *source* $s(e)$ and *target* $t(e)$. The partial order on B captures containment, and thus $b \leq b'$ is interpreted as asserting the

containment of b in b', which trivially includes the case of b and b' being the same blob. The partial order on the edges is of lesser importance for the purposes of the present paper. More broadly, however, it is a justified generalisation over pictorial intuition and, as detailed in [11], also a necessary device in studying completion (Fig. 1(c)) and "conflicts" among the edges. We write $b < b'$ to mean ($b \leq b'$ and $b \neq b'$). Therefore, $b \not< b'$ holds when $b \geq b'$ or when b and b' are unrelated by the partial order. The spatial containment of blobs is commonly referred to as *depth*.

Example 1. Fig. 1(a) may be seen as the pictorial representation of the higraph with blobs $\{A, B, C, D, E, F\}$ where $B, C, D < A$; edges $\{e_1, e_2, e_3, e_4\}$ where $e_2 < e_1$; $s(e_1) = E$, $t(e_1) = A$, $s(e_2) = E$, $t(e_2) = C$, and so on. □

Note, however, that 

and 

are both pictorial representations of the *same* higraph. That is, the definition of higraph does not account for non-trivial intersections [6]. (We say that two blobs b_1 and b_2 in a higraph intersect *trivially* whenever there is a third blob b_3 contained in both b_1 and b_2.) Yet, the need for intersections often arises most naturally in the diagrammatic modelling of computing structures, and particularly of complex and interrelated datasets, as is evident in ER modelling with higraphs [6] and in constraint modelling [3].

Also in computing, higraphs are employed as uncluttered representations of (the graphs which underlie) complex state-transition systems. The basic idea behind such efficient, higraph-based representations is to generalise the notion of transition from individual states to whole *collections of states*. Each such collection, concretely represented as a blob which in turn contains other blobs, corresponds to a (conceptual or actual) *sub-system*.

Example 2. A higher-level edge such as the one from A to F in Fig. 1(a) is understood as implying lower-level edges, in this case from each of the blobs B, C and D contained in the sub-system represented by A, to the target blob F. Thus, the higraph of Fig. 1(a) concisely represents the transition system in Fig. 1(b). □

In general, this relationship of the underlying transition system to its representation as a higraph is understood in terms of *completing* a higraph with the addition of all implied edges. This completion operation, which we characterised mathematically in [11], is illustrated in Fig. 1(c) (in which added edges are shown dashed).

We therefore seek a notion of *dynamics* (i.e. of state-transition behaviour) for higraphs which implicitly contains the behaviour of the represented transition system. Central to the dynamics, as well as to more "static" interpretations, is a notion of a *path* appropriate for higraphs:

Definition 2. *A* path *in a higraph is a finite sequence* $\langle e_0, \ldots, e_{n-1} \rangle$ *of edges such that* $t(e_i) \leq s(e_{i+1})$ *or* $s(e_{i+1}) \leq t(e_i)$*, i.e. the target* $t(e_i)$ *of* e_i *is contained in the source* $s(e_{i+1})$ *of* e_{i+1}*, or vice versa.* □

One of the most fundamental operations on higraphs is that of zooming out [6]. The idea is that one often wants to eliminate all structure contained within a given (i.e. selected) blob in a higraph. More precisely, zooming results in the selected blob becoming *atomic* (or *minimal*) by identifying with it all the blobs it contains. Fig. 1(d) illustrates the result of zooming out of A in the higraph of Fig. 1(a). Because abstraction of subsystems is such an essential device in reasoning about complex systems, and zooming is such a natural way of effecting this kind of abstraction on higraphs, the practical importance of zooming in reasoning is profound.

In order, however, to *soundly* infer results about the dynamics of a higraph from the dynamics of its zoom-out, one must know precisely which paths in latter higraph also exist in former. In other words, it is imperative that any reasonable operation of zooming-out does not introduce paths in a way which may lead to false inferences. So, in Section 6, once the terms in its statement have been precisely defined, we prove a generalisation of the following

Theorem: Whenever a higraph μ, in which blobs may intersect, zooms out to a higraph μ', then every (must-)path in μ' is *reflected* (i.e. is the image of) a (must-)path in μ.

Unfortunately, this requirement is failed by the simple, almost naive notion of zooming-out which we have so far outlined:

Example 3. Observe that zooming has created the path $\langle e_2, e_3 \rangle$ from E to itself in Fig. 1(d) which does not exist in the original higraph of Fig. 1(a). □

In response to this problem, we require a more subtle notion of higraph which permits edges to be loosely attached to blobs. Zooming out of blob A in Fig. 1(a) now results in the *loose higraph* of Fig. 2. Here, the understanding is that the

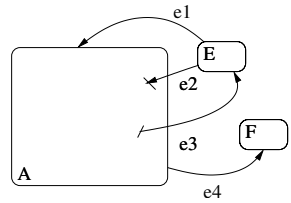


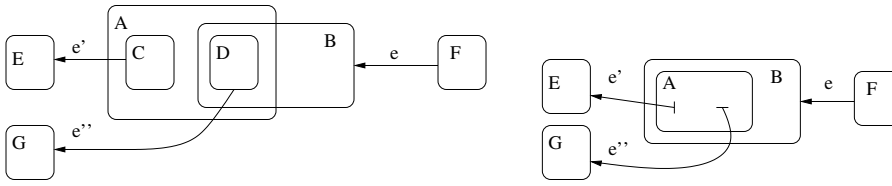
Fig. 2. Zoom-out of the higraph in Fig. 1(a) as a loose higraph.

suppressed target of e_2, indicated as having been some blob originally contained in A, may have not been the same as the suppressed source of e_3. (We note in passing that, should an edge had existed between, say, B and C in Fig.1(a), it would have appeared as a loose edge from A to itself in Fig.2. For the purposes of our semantic analysis, as well as mathematically natural reasons, we retain such completely loose edges in our definitions. To remove them, as implementations of zooming typically do, a further operation may easily be defined and which

composes with our basic notion of zooming.) Thus, in contrast with ordinary higraphs where only one notion of path exists, the introduction of loose edges deliberately creates a distinction between two notions of path: *must-paths* (i.e. certain paths), such as $\langle e_1, e_4 \rangle$ in Fig. 2, and *may-paths*, such as $\langle e_2, e_3 \rangle$. Observe also that $\langle e_1, e_3 \rangle$ is a must-path, as e_1 implies edges to all blobs contained in A, including the suppressed source of e_3. In terms of the above distinction, we consequently demand at least that the theorem above holds of loose higraphs and must-paths.

To obtain this important result in general, a further extension to the notion of (loose) higraph is still required. Problematic situations, which are not resolved by the introduction of loose edges alone, still arise when blobs intersect by means of having common sub-blobs.

Example 4. Fig. 3(a) illustrates a (loose) higraph, where blobs A and B intersect by means of containing D. Zooming out of A in that loose higraph produces,



(a) A (loose) higraph featuring blob intersection.

(b) Problematic zoom-out of blob A in (a)

Fig. 3. The problem of intersecting blobs.

rather unexpectedly, the result of Fig. 3(b), in which a must-path $\langle e, e' \rangle$ has been created that did not previously exist. □

The problem in the preceding example is caused by the inclusion of A into B in Fig. 3(b), which occurs as a side-effect of identifying D with A, owing to the intuitive coherence requirement that zooming must also respect the original containment of D in B. Thus, any reasonable notion of zooming here must not identify D with A (or with B, in the case of zooming out of B). Intuitively, resolving the situation requires the identification of D with a new entity representing the *intersection* of A with B in the picture of Fig. 4. Thus, we have argued that, generally, zooming necessitates an extension to higraphs which substantiates blob intersections.

3 Meet Higraphs

In response, the present paper develops the notion of a *higraph with meets*, and its "loose" variant, together with an appropriate operation of zooming-out. To

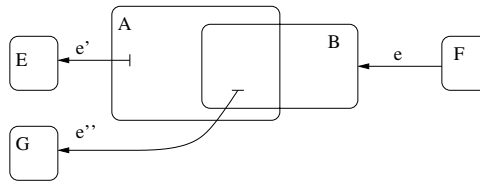


Fig. 4. Intuitive zoom-out of Fig. 3(a).

explicitly account for a notion of intersection among blobs (which, in particular, includes trivial intersections), we first endow the poset B in Definition 1 with meet semi-lattice structure, thus resulting in the following:

Definition 3. *A* meet higraph *consists of: a poset E; a* finite*, $\wedge$-semi-lattice B; and monotone functions $s, t : E \to B$.* □

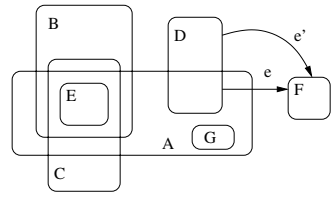
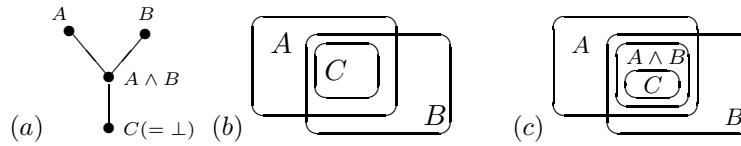


Fig. 5. Meet higraph in which blobs A and D intersect trivially.

Example 5. Fig. 5 represents a meet higraph with underlying $\wedge$-semi-lattice having elements $\{A, B, C, D, E, F, G, A \wedge B, B \wedge C, A \wedge B \wedge C, A \wedge D, \bot\}$, and where $G < A$, $E < A \wedge B \wedge C$, $A \wedge D < A$, and so on; $\bot$ is the least element (which is, by convention, not pictured); $A \wedge G = G$, $F \wedge E = \ldots = F \wedge D \wedge E = \ldots = \bot$ and so on; $s(e) = A \wedge D$ and $s(e') = D$. □

To illustrate the subtle difference between Definitions 1 and 3, the following example compares and contrasts the relation of the two notions of higraph wrt. the usual pictorial representation and intuition:

Example 6. Consider the four-point $\wedge$-semi-lattice shown as a Hasse diagram in (a) below. Regarded as a meet higraph with no edges, it corresponds to the picture shown in (b) below. By contrast, when regarded as an ordinary higraph without edges it corresponds to the picture (c):



Further, to capture the notion of selecting a blob (or, more generally, intersection region) in a meet higraph we introduce the following:

Definition 4. *A* pointed *meet higraph is a meet higraph* $(B, E; s, t : E \to B)$ *together with a distinguished element* p *of* B*, called the* point. □

We have informally argued in Sec. 2 that the effect of zooming on a pointed meet higraph should be to identify each blob strictly contained in the point with the intersection of certain blobs that contain it. In zooming out of A in Fig. 5, for instance, one intuitively expects that G becomes identified with A, and E with $A \wedge B \wedge C$. To make this idea precise, we introduce and study in the next section the salient and topologically natural notion of *hull*.

4 Hulls

Definition 5. *Let* X *be finite,* $\wedge$*-semi-lattice and* $x_0 \in X$*. For every* $x \leq x_0$*, the* hull of x relative to x_0 *is the meet of all elements not strictly less than* x_0 *which are greater than or equal to* x*:*

$$hull_{x_0}(x) \stackrel{\text{def}}{=} \bigwedge \{x' \mid x' \not< x_0 \text{ and } x \leq x'\} \quad \square$$

Example 7. Taking X to be the $\wedge$-semi-lattice underlying the example of Fig. 5 and x_0 to be the blob labelled 'A', one has $hull_A(G) = A$, $hull_A(E) = A \wedge B \wedge C$, $hull_A(A) = A$, $hull_A(A \wedge D) = A \wedge D$ and so on. □

To establish some of the intuition behind Def. 5 and facilitate the understanding of subsequent technical proofs, we record in the following Lemma and its corollary three basic properties of hulls (while deferring their proof to the Appendix).

Lemma 1. *For every finite* $\wedge$*-semi-lattice* X *and* $x, x_0 \in X$ *such that* $x \leq x_0$*:*

1. $x \leq hull_{x_0}(x) \leq x_0$
2. $hull_{x_0}(hull_{x_0}(x)) \leq hull_{x_0}(x)$. □

Corollary 1. *For all* $x \leq x_0$*,* $hull_{x_0}(hull_{x_0}(x)) = hull_{x_0}(x)$. □

The concept of hull is not only central to the more sophisticated notion of zooming which is sought here. Being a topologically natural concept, it is a particularly natural notion to associate not only with meet higraphs, but also with many other notions of graph which feature intersections.

To formalise the connection between hulls and the sought notion of zooming which emerged from the analysis in the previous section, we consider first the simple case of a pointed meet higraph with no edges. Every such meet higraph is, in essence, a pointed, finite, $\wedge$-semi-lattice $\langle B, p\rangle$. Then, in this degenerate case of no edges, the required operation of zooming out of p should have precisely the effect of identifying each $b \leq p$ with $hull_p(b)$. (This may be seen by recalling the transition from Fig. 3(a) to Fig. 4 while ignoring all edges in the two pictures.) The following definition makes this construction explicit:

Definition 6. *Given a pointed, finite $\wedge$-semi-lattice $\langle X, x_0\rangle$, define a function $\zeta_{\langle X,x_0\rangle} : \langle X, x_0\rangle \to \langle X, x_0\rangle$ as follows:*

$$\zeta_{\langle X,x_0\rangle}(x) \stackrel{\text{def}}{=} \begin{cases} hull_{x_0}(x), & \text{if } x \leq x_0 \\ x, & \text{otherwise} \end{cases}$$

Beyond being monotone (Lemma 3, Appendix), each function $\zeta_{\langle X,x_0\rangle}$ inherits, as a corollary of Lemma 1 above, properties which intuitively include idempotency: $\zeta_{\langle X,x_0\rangle} \circ \zeta_{\langle X,x_0\rangle} = \zeta_{\langle X,x_0\rangle}$.

Regarding $\langle B, p\rangle$ as a pointed meet higraph without edges we may now precisely, with the aid of the function $\zeta_{\langle B,p\rangle}$, capture its "zoom-out" $\zeta(\langle B, p\rangle)$, as in the following:

Definition 7. *For every pointed, finite $\wedge$-semi-lattice $\langle X, x_0\rangle$ define $\zeta(\langle X, x_0\rangle)$ to be the pair $\langle X_\zeta, x_0\rangle$, where X_ζ is the sub-poset of X with elements all those $x \in X$ such that $\zeta_{\langle X,x_0\rangle}(x) = x$.* □

Example 8. For $\langle X, x_0\rangle$ as in Example 7, $X_\zeta = X \setminus \{E, G\}$. □

For this construction to be meaningful in our context one must establish that X_ζ is a sub-$\wedge$-semi-lattice of X and that $x_0 \in X_\zeta$, thus making each $\zeta(\langle X, x_0\rangle)$ a finite, pointed $\wedge$-semi-lattice. Indeed, it is not hard to prove the following

Proposition 1. *For every finite, pointed $\wedge$-semi-lattice $\langle X, x_0\rangle$, the poset X_ζ of Definition 7 is a sub-$\wedge$-semi-lattice of X. Moreover, $x_0 \in X_\zeta$, and for all $x \leq x_0$ in X_ζ, $hull_{x_0}(x) = x$.* □

The statement $hull_{x_0}(x) = x$ in the preceding proposition asserts, in agreement with one's intuition about zooming, that the point of $\zeta(\langle B, p\rangle)$ is *minimal*, in the specific sense that all hulls relative to it are trivial. To make this notion of minimality precise before discussing its intuitive appeal, we introduce the following

Definition 8. *The point x_0 in a pointed, finite, $\wedge$-semi-lattice $\langle X, x_0\rangle$ is* minimal *(with respect to hulls) if $hull_{x_0}(x) = x$ for all $x \leq x_0$.* □

The reader is urged to observe the difference between this hull-specific notion of minimality and the usual order-theoretic one ($x \in X$ is order-theoretically minimal in X whenever $x' \leq x \implies x' = x$ for all $x' \in X$). In particular, minimality wrt. hulls does not not imply minimality in the order-theoretic sense. Instead, the notion of minimality in Def. 8 alludes an intuitive notion of "pictorial minimality" expressed in terms of the spatial containment of contours representing blobs in pictures of meet higraphs.

Example 9. Consider again the $\wedge$-semi-lattice underlying the example of Fig. 5. The point corresponding to the blob labelled 'A' in the same figure is *not* minimal wrt. hulls, as $hull_A(E) = A \wedge B \wedge C$. By contrast, the point labelled 'D' *is* minimal wrt. hulls, as $D \wedge A$ is the sole element less than D and $hull_D(D \wedge A) = D \wedge A$. Neither A nor D, however, are minimal in the usual order-theoretic sense. Yet, the contour labelled D in the figure is "pictorially minimal" in the intuitive sense of not wholly containing the contour of any other blob.

Before proceeding to considering edges, we explicate (Theorem 1, below) the universal property associated with Def. 7. It arises from the minimality (wrt. hulls) property of the point in each $\zeta(\langle X, x_0\rangle)$, and will be much useful later.

Definition 9. *Let* $\mathbf{SL}_\star^{fin,\wedge}$ *denote the category having objects all finite, pointed* $\wedge$*-semi-lattices, and arrows* $f : \rightarrow \langle X, x_0\rangle\langle Y, y_0\rangle$ *all monotone functions* $f : \rightarrow XY$ *which, in addition, preserve points* and *hulls: i.e.,* $f(x_0) = y_0$ *and* $\forall\ x.\ x \leq x_0 \implies f(hull_{x_0}(x)) = hull_{y_0}(f(x))$*. Further, let* $\mathbf{SL}_{\star,min}^{fin,\wedge}$ *denote the full subcategory of* $\mathbf{SL}_\star^{fin,\wedge}$ *consisting of all those objects* $\langle X, x_0\rangle$ *in which the point* x_0 *is* minimal *in the sense of Def. 8. We write* J *for the full and faithful functor including* $\mathbf{SL}_{\star,min}^{fin,\wedge}$ *into* $\mathbf{SL}_\star^{fin,\wedge}$*.* □

Noteworthy here for its central role in subsequent development, and for explaining how the elements of each $\langle B, p\rangle$ map to elements in its "zoom-out" $\zeta(\langle B, p\rangle)$, is the following family of morphisms:

Definition 10. *For each finite,* $\wedge$*-semi-lattice* $\langle X, x_0\rangle$ *let* $\eta_{\langle X,x_0\rangle} : \langle X, x_0\rangle \rightarrow J(\zeta(\langle X, x_0\rangle))$ *be the morphism in* $\mathbf{SL}_\star^{fin,\wedge}$ *defined by* $x \in X \mapsto \zeta_{\langle X,x_0\rangle}(x)$*.* □

The statement of the following Theorem is now the expression, in very precise terms, of the intuitive understanding of $\zeta(\langle X, x_0\rangle)$ as obtained from $\langle X, x_0\rangle$ by making the point x_0 minimal wrt. hulls, but without otherwise disturbing the structure of $\langle X, x_0\rangle$:

Theorem 1. *The function* $\zeta : Obj(\mathbf{SL}_\star^{fin,\wedge}) \rightarrow Obj(\mathbf{SL}_{\star,min}^{fin,\wedge})$ *of Def. 7 extends to a functor* $\zeta : \mathbf{SL}_\star^{fin,\wedge} \rightarrow \mathbf{SL}_{\star,min}^{fin,\wedge}$ *which is left adjoint to* J*. The* unit η *of the adjunction has components the morphism of Def. 10.*

Proof. Consider any arrow $f : \langle X, x_0\rangle \rightarrow J(\langle Y, y_0\rangle)$ *in* $\mathbf{SL}_\star^{fin,\wedge}$ *and any* $x \in X$ *such that* $x \leq x_0$*. Since all hulls in* $\langle Y, y_0\rangle$ *are trivial (wrt.* y_0*), it follows from* $f(x) \leq f(x_0) = y_0$ *that* $hull_{y_0}(f(x)) = f(x)$*. Preservation of hulls by* f *now yields:* $f(hull_{x_0}(x)) = hull_{y_0}(f(x)) = f(x)$*. Thus, we have shown that* $\forall\ x.\ x \leq x_0 \implies f(hull_{x_0}(x)) = f(x)$*. From this it follows easily that there exists morphism* $f^\flat : \zeta(\langle X, x_0\rangle) \rightarrow \langle Y, y_0\rangle$ *in* $\mathbf{SL}_{\star,min}^{fin,\wedge}$ *such that* $f = J(f^\flat) \circ \eta_{\langle X,x_0\rangle}$*. Since each* $\eta_{\langle X,x_0\rangle}$ *is epi, this factorisation is unique.* □

5 Loose Meet Higraphs and Zooming-Out

Having introduced hulls and the mechanics of zooming-out in the restricted case of meet higraphs with no edges, we proceed to treat the general case. While, as we argued in Sec. 2, the addition of "loose edges" to ordinary higraphs is in itself insufficient, loose edges are still a necessary device in developing a full solution to the problem of reasoning with the aid of zooming-out abstraction. We therefore begin by allowing edges in our meet higraphs to be loosely attached, thus resulting in a corresponding notion of *loose meet higraph*.

Consider again the loose higraph of Fig. 2. We recall from [2, 12] that every such loose higraph with blobs B can be formally cast as an ordinary higraph

having the same edges but containing two distinct copies $\langle 0, b\rangle$ and $\langle 1, b\rangle$ of each $b \in B$, tagged with 0's and 1's. In the pictorial representation of loose higraphs the convention is that blobs tagged with 0 are not shown at all and that, for instance, an edge with target of the form $\langle 0, b\rangle$, such as e_2 in Fig. 2 with target $\langle 0, A\rangle$, has its endpoint lying *inside* the contour picturing b. Formally,

Definition 11. *Given any poset B define poset $B^\sharp$ as having underlying set $\{0, 1\} \times B$, and partial order generated by the following two rules:*

- $\langle 0, b\rangle < \langle 1, b\rangle$ *for all* $b \in B$*; and*
- $\langle 1, b\rangle \leq \langle 1, b'\rangle$ *whenever* $b \leq_B b'$ *in* B.

Further, define the "projection" $\pi_B : B^\sharp \to B$ as mapping both $\langle 0, b\rangle$ and $\langle 1, b\rangle$ to b, for each $b \in B$. The function $(-)^\sharp$ extends to an endofunctor on **Poset**, *by taking $(f : A \to B)^\sharp$ to be the monotone map sending $\langle i, a\rangle$ to $\langle i, f(a)\rangle$.* □

We shall use $b, b', b_1, \ldots$ to range over the poset B and $v, v', v_1, \ldots$ to range over $B^\sharp$. Using this auxiliary poset structure we can now make precise the definition of a meet higraph with loosely attached edges:

Definition 12. *A* meet higraph with loosely attached edges *(or* loose meet higraph *for short) is a quadruple $(B, E; s, t : E \to B^\sharp)$ where: E is a poset; B is a finite, $\wedge$-semi-lattice; and $s, t : E \to B^\sharp$ are functions making both $\pi_B \circ s$ and $\pi_B \circ t$ monotone. A* pointed, loose meet higraph $\mu_\star = \langle \mu, p\rangle$ *consists of a loose meet higraph $\mu = (B, E; s, t : E \to B^\sharp)$ together with a distinguished element $p \in B$ called the* point. □

Example 10. In the loose meet higraph of Fig. 4, one has $s(e') = \langle 0, A\rangle$, $s(e'') = \langle 0, A \wedge B\rangle$, $t(e) = \langle 1, B\rangle$, and so on. □

We now seek a notion of morphism for pointed loose meet higraphs which smoothly extends the morphisms of Def. 10 to the new setting. In addition to components f_E, f_B which map the elements (edges and blobs) that are visible in pictures, such a morphism f must also have a component $f_{B^\sharp}$ which also maps the invisible elements. While $f_{B^\sharp}$ cannot, in general, be monotone (as $\langle 0, b\rangle \not\leq \langle 0, b'\rangle$ even when $b \leq b'$), it must at least be consistent with f_B and well-behaved with respect to hulls. Making this precise, we have:

Definition 13. *A* morphism $f : \langle \mu_0, p_0\rangle \to \langle \mu_1, p_1\rangle$ *of pointed loose meet higraphs, where $\mu_0 = (B_0, E_0; s_0, t_0 : E_0 \to {B_0}^\sharp)$ and $\mu_1 = (B_1, E_1; s_1, t_1 : E_1 \to {B_1}^\sharp)$, consists of: a monotone function $f_E : E_0 \to E_1$; a monotone and hull-preserving function $f_B : B_0 \to B_1$; and a function $f_{B^\sharp} : {B_0}^\sharp \to {B_1}^\sharp$. These data are subject to the following conditions:*

- *sources and targets of edges are preserved: e.g., $s_1 \circ f_E = f_{B^\sharp} \circ s_0$;*
- *f_B preserves the point and $f_{B^\sharp}$ is consistent with f_B: $f_B \circ \pi_{B_0} = \pi_{B_1} \circ f_{B^\sharp}$;*
- *$f_{B^\sharp}$* reflects *hulls, in the sense that $f_{B^\sharp}(v) = \langle 1, hull_{p_1}(f_B(b))\rangle$ implies $v = \langle 1, hull_{p_0}(b)\rangle$ for all $b \leq p_0$ in B_0 and $v \in {B_0}^\sharp$.* □

Morphisms of pointed, loose higraphs compose component-wise and have obvious identities. Thus one has a category $\mathcal{LMH}_\star$ of pointed loose meet higraphs.

In the last section we described how zooming out of a pointed meet higraph without edges makes the point in $\langle B, p\rangle$ minimal wrt. hulls in $\zeta(\langle B, p\rangle)$, subject to the universal property of Thm. 1. A contrasting look at figures 3(a) and 4 confirms one's intuition that a similar universal property must hold in the general case of zooming out of an arbitrary loose meet higraph $\mu_\star$. This is what we prove in the remainder of this section, starting with defining how the sources and targets of edges in $\mu_\star$ are appropriately fixed in the zoom-out $Z(\mu_\star)$:

Definition 14. *Let $\mu_\star = \langle \mu, p\rangle$ be a pointed loose meet higraph with μ being $(B, E; s, t : E \to B^\sharp)$. Formally, $Z(\mu_\star)$ is the pointed loose meet higraph given by $(B_\zeta\ , E\ ; \eta_{B^\sharp} \circ s\ , \eta_{B^\sharp} \circ t : E\ \to {B_\zeta}^\sharp)$ and point $p \in B_\zeta$, where $\eta_{B^\sharp} : B^\sharp \to {B_\zeta}^\sharp$ is the function sending each $\langle i, b\rangle \in B^\sharp$ such that $b \notin B_\zeta$ to $\langle 0, \eta_{\langle B,p\rangle}(b)\rangle$; and to $\langle i, \eta_{\langle B,p\rangle}(b)\rangle$ otherwise. (Recall that $\eta_{\langle B,p\rangle}$, regarded here as a monotone function from B to B_ζ, is defined in Def 10 and that B_ζ is the $\wedge$-semi-lattice underlying $\zeta(\langle B, p\rangle)$ in Def 7.)* □

Example 11. The preceding definition formalises how the source of an edge such as, say, e'' in Fig. 3(a) is fixed to $\langle 0, A \wedge B\rangle$ in Fig. 4, after its original source $\langle 1, D\rangle$ becomes identified with $\langle 0, \mathit{hull}_A(D)\rangle = \langle 0, A \wedge B\rangle$ as a result of zooming out of blob A. □

With $\mathcal{LMH}_{\star,min}$ we shall denote the (full) subcategory of $\mathcal{LMH}_\star$ consisting of all pointed, loose meet higraphs in which the point is minimal in the sense of Def. 8. Thus the function Z in Def. 14, being a function from the objects of $\mathcal{LMH}_\star$ to those of $\mathcal{LMH}_{\star,min}$, formalises the intuitive understanding of zooming out as reducing the point of $\mu_\star$ to a minimal (wrt. hulls) point in $Z(\mu_\star)$. Moreover, it does so without otherwise disturbing the structure of $\mu_\star$. Before proving this universal property of Z in Thm. 2 below, we need an intuitive mapping (i.e. morphism) from each $\mu_\star$ to its zoom-out:

Definition 15. *Let I be the (fully faithful) inclusion functor $\mathcal{LMH}_{\star,min} \to \mathcal{LMH}_\star$. For each object $\mu_\star = \langle \mu, p\rangle$ of $\mathcal{LMH}_\star$, where $\mu = (B, E; s, t : E \to B^\sharp)$, define a morphism $\eta_{\mu_\star} : \mu_\star \to I(Z(\mu_\star))$ with the following components: $(\eta_{\mu_\star})_E = \mathrm{id}_E : E \to E$; $(\eta_{\mu_\star})_B = \eta_{\langle B,p\rangle}$ (where $\eta_{\langle B,p\rangle}$ is the morphism of Def 10 regarded here as a monotone, hull preserving function from B to B_ζ) ; and $(\eta_{\mu_\star})_{B^\sharp}$ is the function $\eta_{B^\sharp}$ in Def. 14.* □

Theorem 2. *The function $Z : Obj(\mathcal{LMH}_\star) \to Obj(\mathcal{LMH}_{\star,min})$ extends to a functor which is left adjoint to I, with unit being the morphisms $\eta_{\mu_\star}$ of Def. 15.*

Proof. (Sketch) Given any morphism $f : \mu_\star \to I(\mu'_\star)$, where $\mu_\star$ and $\mu'_\star$ are objects of $\mathcal{LMH}_\star$ and $\mathcal{LMH}_{\star,min}$ respectively, define a morphism $f^\flat : Z(\mu_\star) \to \mu'_\star$ in $\mathcal{LMH}_{\star,min}$ as having the following components: $f^\flat_E$ maps each $e \in E$ to $f_E(e)$; $f^\flat_B$ is the induced by the adjunction in Thm. 1, unique morphism $f^\flat_{\langle B,p\rangle} : B \to B'$

such that $f^\flat_B \circ (\eta_{\mu_\star})_B = f_B$*; and* $f^\flat_{B^\sharp} = f_{B^\sharp} \circ \iota_{\langle B,p\rangle}$ *where* $\iota_{\langle B,p\rangle}$ *is the inclusion of* $\zeta(\langle B,p\rangle) = \langle B_\zeta, p\rangle$ *into* $\langle B,p\rangle$*.*

In addition to $f^\flat_E \circ (\eta_{\mu_\star})_E = f_E$ *and* $f^\flat_B \circ (\eta_{\mu_\star})_B = f_B$*, which hold straightforwardly, one also has (Lemma 5, Appendix) that* $f^\flat_{B^\sharp} \circ (\eta_{\mu_\star})_{B^\sharp} = f_{B^\sharp}$*. Thus,* $f = I(f^\flat) \circ \eta_{\mu_\star}$*. As each component of* $\eta_{\mu_\star}$ *is epi,* $f^\flat$ *is moreover the unique morphism in* $\mathcal{LMH}_{\star,min}$ *with this property, and the proof concludes with an appeal to Theorem IV.2(ii) of [7].* □

The preceding result directly parallels the corresponding theorem in [12] for (non-meet) loose higraphs. What, however, the new notion of zooming on loose meet higraphs makes possible, over and above previous work in [12], is the ability to fully relate the semantics before and after a zoom-out, as we show next.

6 Dynamics of Loose Meet Higraphs

In this section we make precise the dynamics, i.e. transition semantics, of loose meet higraphs by introducing a notion of *run* akin to similar notions in use with ordinary ("flat") transition systems. In an ordinary higraph, or a meet higraph, a run is essentially a sequence of edges (transitions) together with the blobs (or *states*, in the popular jargon of most applications) which are traversed by performing the transitions:

Definition 16. *A* run *through a meet higraph* $(B, E; s, t : E \to B)$ *is a finite sequence of the form* $b_0 \xrightarrow{e_1} b_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} b_n$ *where, for all* $1 \leq i \leq n$*,* $b_{i-1} \leq_B s(e_i)$ *and* $b_i \leq_B t(e_i)$*.* □

Example 12. In Fig. 1(a) (regarded trivially as a meet higraph), $E \xrightarrow{e_1} A \xrightarrow{e_4} F$ is clearly a run, as is $E \xrightarrow{e_1} D \xrightarrow{e_3} E$ because $s(e_3) = D < A = t(e_1)$. Similarly, $E \xrightarrow{e_2} C \xrightarrow{e_4} F$ is also a run because $t(e_2) = C < A = s(e_4)$. □

In ordinary transition systems, the sequence of states traversed is implicit in the notion of path (i.e. connected sequence of transitions). In higraphs and meet higraphs, where higher-level edges are taken to imply lower-level ones, as is implicit in the joining condition $b_{i-1} \leq_B s(e_i)$ and $b_i \leq_B t(e_i)$ of Def. 16 above, the notion of path no longer provides adequate state information, hence the need for a notion of "run". Suggestive of dynamics though its name might be, we wish to stress that the mathematical structure of runs also pertains, by subsuming the notion of path, to more static interpretations of higraphs.

Recall (Section 2) how the introduction of loosely attached edges incurs a distinction between *must-* and *may-paths*. Intuitively, the idea is that a may-path may perish, in the sense of becoming disconnected, when one makes explicit the suppressed end-points of every loose edge in the path. In Fig. 2, for instance, one may introduce two new, distinct blobs C and D contained in A, so as to make $t(e_2) = \langle 1, C\rangle$ and $s(e_3) = \langle 1, D\rangle$, thereby making $\langle e_2, e_3\rangle$ a non-path. By

contrast, a must-path must persist, no matter how one attaches its loose edges to newly introduced blobs.

A similar distinction between *must-run*s and *may-run*s through a loose meet higraph is therefore necessary. Here we concentrate on must-runs only:

Definition 17. *A* must-run *through a pointed, loose meet higraph $\mu_\star$, where μ is $(B, E; s, t : E \to B^\sharp)$, is a sequence of the form $\langle i_0, b_0\rangle \xrightarrow{e_1} \langle i_1, b_1\rangle \xrightarrow{e_2} \ldots \xrightarrow{e_n} \langle i_n, b_n\rangle$, where $\langle i_j, b_j\rangle \in B^\sharp$ and $e_j \in E$, subject to the following conditions:*

1. *for all $1 \leq j \leq n$, $\langle i_j, b_j\rangle \leq t(e_j)$ and $\langle i_{j-1}, b_{j-1}\rangle \leq s(e_j)$; and*
2. *$\pi_0(t(e_j)) = 1$ or $\pi_0(s(e_{j+1})) = 1$ for all $1 \leq j < n$, where π_0 is the projection mapping each $\langle i, b\rangle \in B^\sharp$ to i.* □

Example 13. Consider Fig. 2, this time as representing a (pointed) loose meet higraph. While $\langle 1, E\rangle \xrightarrow{e_1} \langle 1, A\rangle \xrightarrow{e_4} \langle 1, F\rangle$ is a run, the sequence $\langle 1, E\rangle \xrightarrow{e_1} \langle 1, A\rangle \xrightarrow{e_3} \langle 1, E\rangle$ isn't because it violates the first condition in Def. 17. However $\langle 1, E\rangle \xrightarrow{e_1} \langle 0, A\rangle \xrightarrow{e_3} \langle 1, E\rangle$ is a run, as is $\langle 1, E\rangle \xrightarrow{e_1} \langle 0, A\rangle \xrightarrow{e_4} \langle 1, F\rangle$. But $\langle 1, E\rangle \xrightarrow{e_2} \langle 0, A\rangle \xrightarrow{e_3} \langle 1, E\rangle$ is not, as it violates the second condition. □

The following theorem now establishes precisely how each must-run through $Z(\mu_\star)$ is the image of a corresponding must-run through $\mu_\star$:

Theorem 3. *Every must-run through $I(Z(\mu_\star))$, where μ is $(B, E; s, t : E \to B^\sharp)$, is of the form $\eta_{\mu_\star}(v_0) \xrightarrow{e_1} \eta_{\mu_\star}(v_1) \xrightarrow{e_2} \ldots \xrightarrow{e_n} \eta_{\mu_\star}(v_n)$, for some run $v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} v_n$ through $\mu_\star$, where $\eta_{\mu_\star}$ is the component at $\mu_\star$ of the unit of the adjunction of Theorem 2.*

Proof. (Sketch) Let $\hat{v_0} \xrightarrow{e_1} \hat{v_1} \xrightarrow{e_2} \ldots \xrightarrow{e_n} \hat{v_n}$, where $\hat{v_j} \in {B_\zeta}^\sharp$, be an arbitrary run through $Z(\mu_\star)$. Take $v_0 = \langle i_0, \pi_B(s(e_1))\rangle$. Calculation establishes $v_0 \leq s(e_1)$ and $\eta_{\mu_\star}(v_0) = \hat{v_0}$. Similarly take $v_n = \langle i_n, \pi_B(t(e_n))\rangle$. For $1 \leq j < n$, each v_j is obtained by an application of Lemma 6 in the Appendix. □

7 Conclusions

Of particular importance to practitioners is the kind of semi-formal, tool-assisted reasoning which consists of the progressive simplification of a diagram, by repeatedly using abstraction, until either a counter-example is reached or the property can easily be proven. The potential for error in this style of reasoning is great, and so supporting tools must intervene to prohibit any unsound steps or inferences. To do so, we have argued using higraphs as concrete examples, tools must often maintain internal representations (such as meet higraphs) which are more sophisticated than the user's notation. Also, such tools must know how the semantics of a zoom-out relates to the semantics of the original diagram. Here we have developed such a semantics and precise relationship based on, but not limited to, the common interpretation of higraphs as transition systems. More generally, we have argued, our work also applies to other notations, particularly those which feature intersections among vertices.

References

[1] R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. In *Symposium on Principles of Programming Languages*, pages 390–402, 2000.

[2] Stuart Anderson, John Power, and Konstantinos Tourlas. Reasoning in higraphs with loose edges. In *Proceedings of the 2001 IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 23–29. IEEE Computer Society Press, September 2001.

[3] J. Gil, J. Howse, and S. Kent. Towards a formalisation of constraint diagrams. In *Proceedings of the 2001 IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 72–79. IEEE Computer Society Press, September 2001.

[4] R. Grosu, Gh. Stefănescu, and M. Broy. Visual formalisms revisited. In *Proc. IEEE Int. Conf. Application of Concurrency to System Design, CSD*, 1998.

[5] David Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8(3):231–275, 1987.

[6] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[7] Saunders MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.

[8] Z. Manna and A. Pnueli. Temporal verification diagrams. In *Proceedings of TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 726–765. Springer-Verlag, 1994.

[9] F. Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *Proceedings of the IEEE Workshop on Visual Languages*, 1991.

[10] Bonnie M. Nardi. *A Small Matter of Programming: Perspectives on End-User Computing*. MIT Press, 1993.

[11] John Power and Konstantinos Tourlas. An algebraic foundation for higraphs. In L. Fribourg, editor, *Proceedings of the 15th Annual Conference of the European Association for Computer Science Logic (CSL)*, volume 2142 of *Lecture Notes in Computer Science*, pages 145–159. Springer-Verlag, September 2001.

[12] John Power Stuart Anderson and Konstantinos Tourlas. Zooming-out on higraph-based diagrams: syntactic and semantic issues. In James Harland, editor, *Electronic Notes in Theoretical Computer Science*, volume 61. Elsevier Science Publishers, 2002.

Appendix: Technical Lemmas and Proofs

Meet Higraphs and Hulls

The following are simple properties arising from the definitions of hull (Def. 5) and of the functions $\zeta_{\langle X,x_0\rangle}$ (Def. 6):

Lemma 1. For every finite $\wedge$-semi-lattice X and $x, x_0 \in X$ such that $x \leq x_0$, the following hold:

1. $x \leq hull_{x_0}(x) \leq x_0$
2. $hull_{x_0}(hull_{x_0}(x)) \leq hull_{x_0}(x)$.

Proof. Fix arbitrary $x \leq x_0$.

- To show (1), observe that $x \leq \bigwedge\{x' \mid x' \not\leq x_0 \text{ and } x \leq x'\}$ because x is a lower bound of that set. Moreover, one clearly has $x_0 \in \{x' \mid x' \not\leq x_0 \text{ and } x \leq x'\}$, and thus $hull_{x_0}(x) \leq x_0$.
- For showing (2) observe that, for all x' such that $x' \not\leq x_0$ and $x \leq x'$, one has $hull_{x_0}(x) \leq x'$ (by the very definition of $hull_{x_0}(x)$ as a meet). Hence, the following inclusion of sets holds: $\{x' \mid x' \not\leq x_0 \text{ and } x \leq x'\} \subseteq \{x' \mid x' \not\leq x_0 \text{ and } hull_{x_0}(x) \leq x'\}$, from which one obtains $hull_{x_0}(hull_{x_0}(x)) \leq hull_{x_0}(x)$.

Lemma 2. *Given any elements x_1, x_2, x_0 in a finite $\wedge$-semi-lattice such that $x_1 \leq x_2 \leq x_0$, one has $hull_{x_0}(x_1) \leq hull_{x_0}(x_2)$.*

Lemma 3. *Each function $\zeta_{\langle X,x_0\rangle}$ is monotone.*

Proof. Arbitrarily fix $x_1, x_2 \in X$ such that $x_1 \leq x_2$. As one always has $x_2 \leq \zeta_{\langle X,x_0\rangle}(x_2)$ (by Lemma 1(1) and Definition 6), we show that $\zeta_{\langle X,x_0\rangle}(x_1) \leq x_2$ by case analysis:

1. *$x_1, x_2 \leq x_0$. Monotonicity follows from Lemma 2 above.*
2. *($x_1 \leq x_0$ and $x_2 \not\leq x_0$). In this case $hull_{x_0}(x_1) \leq x_2$ because x_2 is an element of the set $\{x' \mid x' \not\leq x_0 \text{ and } x_1 \leq x'\}$. It follows from Definition 6 that $\zeta_{\langle X,x_0\rangle}(x_1) = hull_{x_0}(x_1) \leq x_2$.*
3. *$x_1 \not\leq x_0$ and $x_2 \not\leq x_0$. In this case, trivially, $\zeta_{\langle X,x_0\rangle}(x_1) = x_1 \leq x_2$.*

Lemmas Used in the Proof of Theorem 2

Lemma 4. *Let $f : \langle \mu_0, p_0\rangle \to \langle \mu_1, p_1\rangle$ be a morphism in $\mathcal{LMH}_\star$, where $\langle \mu_1, p_1\rangle$ is minimally pointed wrt. hulls. Then, for all $b \leq p_0$ the following two hold:*

1. *$f_{B^\sharp}(\langle 0, b\rangle) = \langle 0, f_B(b)\rangle$; and*
2. *$b \notin B_\zeta \implies f_{B^\sharp}(\langle 1, b\rangle) = \langle 0, f_B(b)\rangle$.*

Proof. From $b \leq p_0$, monotonicity of f_B and preservation of points by f_B it follows that $f_B(b) \leq p_1$, whereby, from the minimality (wrt. hulls) of p_1, one has

$$f_B(b) = hull_{p_1}(f_B(b)) \ . \tag{1}$$

To show (1) assume $f_{B^\sharp}(\langle 0, b\rangle) \neq \langle 0, f_B(b)\rangle$ and derive a contradiction. The only possibility is $f_{B^\sharp}(\langle 0, b\rangle) = \langle 1, f_B(b)\rangle$ whereby, from equation (1) and condition (3) of Def. 13, it follows that $\langle 0, b\rangle = \langle 1, hull_{p_0}(b)\rangle$: a contradiction, as $0 \neq 1$.

To show the contrapositive of (2) assume $f_{B^\sharp}(\langle 1, b\rangle) \neq \langle 0, f_B(b)\rangle$. The only possibility is $f_{B^\sharp}(\langle 1, b\rangle) = \langle 1, f_B(b)\rangle$ whereby, from equation (1) and condition (3) of Def. 13, it follows that $\langle 1, b\rangle = \langle 1, hull_{p_0}(b)\rangle$, eqivalently $b = hull_{p_0}(b)$, equivalently $b \in B_\zeta$ by Def 7.

Lemma 5. *Let $f^\flat_{B^\sharp}$, $(\eta\mu_\star)_{B^\sharp}$ and $f_{B^\sharp}$ be as in the sketched proof of Thm 2. Then $f^\flat_{B^\sharp} \circ (\eta_{\mu_\star})_{B^\sharp} = f_{B^\sharp}$.*

Proof. For notational simplicity, we abbreviate $\eta_{\mu_\star}$ to just η in the course of this proof. Consider now arbitrary $\langle i, b\rangle \in B^\sharp$.

- *In the case of $b \in B_\zeta$, the equation $(f^\flat_{B^\sharp} \circ \eta_{B^\sharp})(\langle i, b\rangle) = f_{B^\sharp}(\langle i, b\rangle)$ is established by straightforward calculation from the definitions of f, η, ι and $f^\flat$.*
- *Assume now that $b \notin B_\zeta$. Then, by the definition of η (recall also Def. 6), it must be that $b \leq p$, where p the point of $\mu_\star$ in Thm. 2. One now has $(f^\flat_{B^\sharp} \circ \eta_{B^\sharp})(\langle i, b\rangle) = f^\flat_{B^\sharp}(\eta_{B^\sharp}(\langle i, b\rangle)) = f^\flat(\langle 0, \eta_B(b)\rangle)$ by definition of η. Further, by two applications of Lemma 4 above (to $f^\flat$ and f respectively), it follows that $f_{B^\sharp}(\langle 0, b\rangle) = \langle 0, f_B(b)\rangle = \langle 0, f^\flat_B(\eta_B(b))\rangle = f^\flat_{B^\sharp}(\langle 0, \eta_B(b)\rangle)$ and thus that*

$$(f^\flat_{B^\sharp} \circ \eta_{B^\sharp})(\langle i, b\rangle) = f_{B^\sharp}(\langle 0, b\rangle) \ . \tag{2}$$

In the case of $i = 0$, equation (2) immediately yields the desired result. When $i = 1$, on the other hand, $b \notin B_\zeta$ and Lemma 4(2) above yield $f_{B^\sharp}(\langle 1, b\rangle) = \langle 0, f_B(b)\rangle = (f^\flat_{B^\sharp} \circ \eta_{B^\sharp})(\langle 1, b\rangle)$ because of equation (2) above.

Lemmas Used in the Proof of Theorem 3

Lemma 6. *Let $\mu_\star$ be $(B, E; s, t : E \to B^\sharp)$ with point p; $e_1, e_2 \in E$ and $\hat{v} \in {B_\zeta}^\sharp$. If $\xrightarrow{e_1} \hat{v} \xrightarrow{e_2}$ is part of a must-run through $Z(\mu_\star)$ (i.e. $\hat{v}$, the target $t'(e_1)$ of e_1 in $Z(\mu_\star)$, and the source $s'(e_2)$ of e_2 in $Z(\mu_\star)$ satisfy the conditions in Def. 17), then there exists $v \in B^\sharp$ such that $\eta_{\mu_\star}(v) = \hat{v}$ and $\xrightarrow{e_1} v \xrightarrow{e_2}$ is part of a must-run through $\mu_\star$.*

Proof. Having arbitrarily fixed $\mu_\star$, we abbreviate as just η the morphism $\eta_{\mu_\star}$ of Def. 15. We also write, for brevity, t_B and s_B for the composites $\pi_B \circ t$ and $\pi_B \circ s$ respectively. Recalling from the definition of $Z(\mu_\star)$ (Def. 14) that $s' = \eta_{B^\sharp} \circ s$ and $t' = \eta_{B^\sharp} \circ t$, we proceed by case analysis:

1. *Case $\pi_0(t'(e_1)) = 1$. Then, by the definitions of t' and $\eta_{B^\sharp}$, one must have $t(e_1) = t'(e_1)$. If also $\pi_0(s'(e_2)) = 1$ then, similarly, $s(e_2) = s'(e_2)$ and one can simply take $v = \hat{v}$. If, on the other hand, one assumes $\pi_0(s'(e_2)) = 0$ then, from condition $\hat{v} \leq s'(e_2)$ and Def. 11, it follows that one must also have $\hat{v} = s'(e_2)$. Now, from condition $\hat{v} \leq t'(e_1)$, it follows that one must have $s'(e_2) \leq t'(e_1)$ which, in turn, implies $\eta_B(s_B(e_2)) \leq \eta_B(t_B(e_2))$. Taking $v = \langle 0, s_B(e_2)\rangle$ one therefore immediately has $v \leq s(e_2)$, as required. Further, $v \leq \langle 1, s_B(e_2)\rangle \leq \langle 1, \eta_B(s_B(e_2))\rangle \leq \langle 1, \eta_B(t_B(e_1))\rangle = t'(e_1) = t(e_1)$ because of the definition of η_B (Def 10), and Def. 6. Finally, $\eta_{B^\sharp}(v) = \eta_{B^\sharp}(\langle 0, s_B(e_2)\rangle) = \langle 0, \eta_B(s_B(e_2))\rangle = s'(e_2) = \hat{v}$.*
2. *Case $\pi_0(t'(e_1)) = 0$. It follows from Def. 17 that one must have $\pi_0(s'(e_2)) = 1$. In that case one takes $v = \langle 0, t_B(e_1)\rangle$ and the proof proceeds in a manner similar to that of the previous case above.*

Deriving Bisimulation Congruences: 2-Categories Vs Precategories *

Vladimiro Sassone[1] and Paweł Sobociński[2]

[1] University of Sussex
[2] University of Aarhus

Abstract. G-relative pushouts (GRPOs) have recently been proposed by the authors as a new foundation for Leifer and Milner's approach to deriving labelled bisimulation congruences from reduction systems. This paper develops the theory of GRPOs further, arguing that they provide a simple and powerful basis towards a comprehensive solution. As an example, we construct GRPOs in a category of 'bunches and wirings.' We then examine the approach based on Milner's precategories and Leifer's functorial reactive systems, and show that it can be recast in a much simpler way into the 2-categorical theory of GRPOs.

Introduction

It is increasingly common for foundational calculi to be presented as *reduction systems*. Starting from their common ancestor, the λ calculus, most recent calculi consist of a reduction system together with a contextual equivalence (built out of basic observations, viz. barbs). The strength of such an approach resides in its intuitiveness. In particular, we need not invent labels to describe the interactions between systems and their possible environments, a procedure that has a degree of arbitrariness (cf. early and late semantics of the π calculus) and may prove quite complex (cf. [5, 4, 3, 1]).

By contrast, reduction semantics suffer at times by their lack of compositionality, and have complex semantic theories because of their contextual equivalences. Labelled bisimulation congruences based on *labelled transition systems* (LTS) may in such cases provide fruitful proof techniques; in particular, bisimulations provide the power and manageability of coinduction, while the closure properties of congruences provide for compositional reasoning.

To associate an LTS with a reduction system involves synthesising a compositional system of labels, so that silent moves (or τ-actions) reflect the original reductions, labels describe potential external interactions, and all together they yield a LTS bisimulation which is a congruence included in the original contextual reduction equivalence. Proving bisimulation is then enough to prove reduction equivalence.

Sewell [19] and Leifer and Milner [13, 11] set out to develop a theory to perform such derivations using general criteria; a meta-theory of *deriving bisimulation congruences*. The basic idea behind their construction is to use contexts as labels. To exemplify the idea, in a CCS-like calculus one would for instance derive a transition

* Research supported by '**DisCo**: Semantic Foundations of Distributed Computation', EU IHP 'Marie Curie' contract HPMT-CT-2001-00290, and **BRICS**, Basic Research in Computer Science, funded by the Danish National Research Foundation.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 409–424, 2003.

$$a.P \xrightarrow{-|\bar{a}.Q} P \mid Q$$

because term $a.P$ in context $- \mid \bar{a}.Q$ reacts to become $P \mid Q$; in other words, the context is a trigger for the reduction.

The first hot spot of the theory is the selection of the right triggers to use as labels. The intuition is to take only the "*smallest*" contexts which allow a given reaction to occur. As well as reducing the size of the LTS, this often makes the resulting bisimulation equivalence finer. Sewell's method is based on dissection lemmas which provide a deep analysis of a term's structure. A generalised, more scalable approach was later developed in [13], where the notion of "smallest" is formalised in categorical terms as a *relative-pushout* (RPOs). Both theories, however, do not seem to scale up to calculi with non trivial *structural congruences*. Already in the case of the monoidal rules that govern parallel composition things become rather involved.

The fundamental difficulty brought about by a structural congruence $\equiv$ is that working up to $\equiv$ gives up too much information about terms for the RPO approach to work as expected. RPOs do not usually exist in such cases, because the fundamental indication of exactly which occurrences of a term constructor belong to the redex becomes blurred. A very simple, yet significant example of this is the category **Bun** of bunch contexts [13], and the same problems arise in structures such as action graphs [14] and bigraphs [15].

In [17] we therefore proposed a framework in which term structure is not explicitly quotiented, but the commutation of diagrams (i.e. equality of terms) is taken up to $\equiv$. Precisely, to give a commuting diagram $rp \equiv sq$ one exhibits a proof α of structural congruence, which we represent as a 2-cell (constructed from the rules generating $\equiv$ and closed under all contexts).

$$\begin{array}{ccc} k & \xrightarrow{p} & l \\ {\scriptstyle q}\downarrow & \alpha & \downarrow{\scriptstyle r} \\ m & \xrightarrow[s]{} & n \end{array}$$

Since such proofs are naturally isomorphisms, we were led to consider G-categories, i.e., 2-categories where all 2-cells are iso, and initiated the study of G-relative pushouts (GRPOs), as a suitable generalisation of RPOs from categories to G-categories.

The purpose of this paper is to continue the development of the theory of GRPOs. We aim to show that, while replacing RPOs at little further complication (cf. §1 and §2), GRPOs significantly advance the field by providing a convenient solution to simple, yet important problems (cf. §3 and §4). The theory of GRPOs promises indeed to be a natural foundation for a meta-theory of 'deriving bisimulation congruences.'

This paper presents two main technical results in support of our claims. Firstly, we prove that the case of the already mentioned category **Bun** of bunch contexts, problematic for RPOs, can be treated in a natural way using GRPOs. Secondly, we show that the notions of precategory and functorial reactive system can be dispensed with in favour of a simpler GRPO-based approach.

The notion of *precategory* is proposed in [11, 12] to handle the examples of Leifer in [11], Milner in [15] and, most recently, of Jensen and Milner in [7]. It consists of a

category appropriately decorated by so-called "*support sets*" which identifies syntactic elements so as to keep track of them under arrow composition. Alas, such supported structures are no longer categories – arrow composition is partial – which makes the theory laborious, and bring us away from the well-known world of categories and their theory. The intensional information recorded in precategories, however, allows one to generate a category "above" where RPOs exist, as opposed to the category of interest "below", say $\mathbb{C}$, where they do not. The category "above" is related to $\mathbb{C}$ via a well-behaved functor, used to map RPOs diagrams from the category "above" to $\mathbb{C}$, where constructing them would be impossible. These structures take the name of *functorial reactive systems*, and give rise to a theory to generate a labelled bisimulation congruences developed in [11].

The paper presents a technique for mapping precategories to G-categories so that the LTS generated using GRPOs is the same as the LTS generated using the above mentioned approach. The translation derives from the precategory's support information a notion of homomorphism, specific to the particular structure in hand, which constitutes the 2-cells of the derived G-category. We claim that this yields an approach mathematically more elegant and considerably simpler than precategories; besides generalising RPOs directly, GRPOs seem to also remove the need for further notions.

Structure of the paper. In §1 we review definitions and results presented in [17]; §2 shows that, analogously to the 1-dimensional case, trace and failures equivalence are congruences provided that enough GRPOs exist. In §3, we show that the category of bunch contexts is naturally a 2-category where GRPOs exist; §4 shows how precategories are subsumed by our notion of GRPOs. Most proofs in this extended abstract are either omitted or sketched. For these, the interested reader should consult [18].

1 Reactive Systems and GRPOs

Lawvere theories [10] provide a canonical way to recast term algebras as categories. For Σ a signature, the (free) Lawvere theory on Σ, say $\mathbf{C}_\Sigma$, has the natural numbers for objects and a morphism $t\colon m \to n$, for t a n-tuple of m-holed terms. Composition is substitution of terms into holes.

Generalising from term rewriting systems on $\mathbf{C}_\Sigma$, Leifer and Milner formulated a definition of *reactive system* [13], and defined a technique to extract labelled bisimulation congruences from them. In order to accommodate calculi with non trivial structural congruences, as explained in the Introduction, we refine their approach as follows.

Definition 1.1. A ***G-category*** is a 2-category where all 2-cells are isomorphisms.

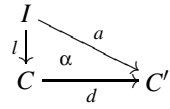
A G-category is a thus a category enriched over G, the category of groupoids.

Definition 1.2. A ***G-reactive system*** $\mathbf{C}$ consists of a G-category $\mathbb{C}$; a subcategory $\mathbb{D}$ of *reactive contexts*, required to be closed under 2-cells and composition-reflecting; a distinguished object $I \in \mathbb{C}$; a set of pairs $\mathcal{R} \subseteq \bigcup_{C\in\mathbb{C}} \mathbb{C}(I,C) \times \mathbb{C}(I,C)$, called the *reaction rules*.

The reactive contexts are those contexts inside which evaluation may occur. By composition-reflecting we mean that $dd' \in \mathbb{D}$ implies d and $d' \in \mathbb{D}$, while the closure property means that given $d \in \mathbb{D}$ and $\rho\colon d \Rightarrow d'$ in $\mathbb{C}$ implies $d' \in \mathbb{D}$. The reaction relation $\longrightarrow\!\!\triangleright$ is defined by taking

$$a \longrightarrow\!\!\triangleright\, dr \quad \text{if there exists } \langle l, r\rangle \in \mathcal{R},\ d \in \mathbb{D} \text{ and } \alpha\colon dl \Rightarrow a \text{ in } \mathbb{C}$$

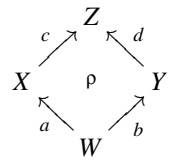
As illustrated by the diagram below, this represents the fact that, up to structural congruence, a is the left-hand side l of a reduction rule in a reaction context d.



The notion of GRPO formalises the idea of a context being the "smallest" that enables a reaction in a G-reactive system, and is a conservative 2-categorical extension of Leifer and Milner RPOs [13] (cf. [17] for a precise comparison).

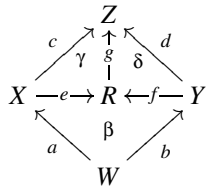
For readers acquainted with 2-dimensional category theory (cf. [9] for a thorough introduction), GRPOs are defined in Definition 1.3. This is followed by an elementary presentation in Proposition 1.4 taken from [17]. We use $\bullet$ for vertical composition.

Definition 1.3 (GRPOs). Let $\rho\colon ca \Rightarrow db\colon W \to Z$ be a 2-cell (see diagram below) in a G-category $\mathbb{C}$. A G-***relative pushout*** (GRPO) for ρ is a bipushout (cf. [8]) of the pair or arrows $(a, \mathbf{1}) : ca \to c$ and $(b, \rho) : ca \to d$ in the pseudo-slice category $\mathbb{C}/Z$.

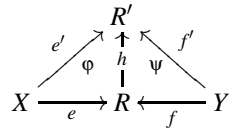


(1)

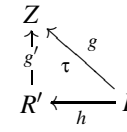
Proposition 1.4. Let $\mathbb{C}$ be a G-category. A candidate GRPO for $\rho\colon ca \Rightarrow db$ as in diagram (1) is a tuple $\langle R, e, f, g, \beta, \gamma, \delta\rangle$ such that $\delta b \bullet g\beta \bullet \gamma a = \rho$ – cf. diagram (*i*).



(*i*)



(*ii*)



(*iii*)

A GRPO for ρ is a candidate which satisfies a universal property. Namely, for any other candidate $\langle R', e', f', g', \beta', \gamma', \delta'\rangle$ there exists a quadruple $\langle h, \varphi, \psi, \tau\rangle$ where $h\colon R \to R'$, $\varphi\colon e' \Rightarrow he$ and $\psi\colon hf \Rightarrow f'$ – cf. diagram (*ii*) – and $\tau\colon g'h \Rightarrow g$ – diagram (*iii*) – which makes the two candidates compatible in the obvious way, i.e.

$$\tau e \bullet g'\varphi \bullet \gamma' = \gamma \qquad \delta' \bullet g'\psi \bullet \tau^{-1} f = \delta \qquad \psi b \bullet h\beta \bullet \varphi a = \beta'.$$

Such a quadruple, which we shall refer to as *mediating morphism*, must be *essentially unique*. Namely, for any other mediating morphism $\langle h', \varphi', \psi', \tau' \rangle$ there must exist a *unique* two cell $\xi\colon h \to h'$ which makes the two mediating morphisms compatible, i.e.

$$\xi e \bullet \varphi = \varphi' \qquad \psi \bullet \xi^{-1} f = \psi' \qquad \tau' \bullet g' \xi = \tau.$$

Observe that whereas RPOs are defined up to isomorphism, GRPOs are defined up to equivalence (since they are bicolimits).

The definition below plays an important role in the following development.

Definition 1.5 (GIPO). Diagram (1) of Definition 1.3 is said to be a G-***idem-pushout*** (GIPO) if $\langle Z, c, d, \mathrm{id}_Z, \rho, \mathbf{1}_c, \mathbf{1}_d \rangle$ is its GRPO.

We recall in §A the essential properties of GRPOs and GIPOs from [17].

Definition 1.6 (LTS). For $\mathbf{C}$ a G-reactive system whose underlying category $\mathbb{C}$ is a G-category, define $\mathsf{GTS}(\mathbf{C})$ as follows:

- the states $\mathsf{GTS}(\mathbf{C})$ are iso-classes of arrows $[a]\colon I \to X$ in $\mathbf{C}$;
- there is a transition $[a] \xrightarrow{[f]} [a']$ if there exists a 2-cell ρ, a rule $\langle l, r \rangle \in \mathcal{R}$, and $d \in \mathbb{D}$ with $a' \cong dr$ and such that the diagram below is a GIPO.

$$\begin{array}{ccccc} & & Z & & \\ & {\scriptstyle f}\nearrow & & \nwarrow{\scriptstyle d} & \\ X & & \rho & & Y \\ & {\scriptstyle a}\nwarrow & & \nearrow{\scriptstyle l} & \\ & & I & & \end{array} \tag{2}$$

Henceforward we shall abuse notation and leave out the square brackets when writing transitions; ie. we shall write simply $a \xrightarrow{f} a'$ instead of $[a] \xrightarrow{[f]} [a']$.

Categories can be seen as a discrete G-categories (the only 2-cells are identities). Using this observation, each G-concepts introduced above reduces to the corresponding 1-categorical concept. For instance, a GRPO in a category is simply a RPO.

2 Congruence Results for GRPOs

The fundamental property that endows the LTS derived from a reduction system with a bisimulation which is a congruence is the following notion.

Definition 2.1 (Redex GRPOs). A G-reactive system $\mathbf{C}$ is said to ***have redex GRPOs*** if every square (2) in its underlying G-category $\mathbb{C}$ with l the left-hand side of a reaction rule $\langle l, r \rangle \in \mathcal{R}$, and $d \in \mathbb{D}$ has a GRPO.

In particular, the main theorem of [17] is as follows.

Theorem 2.2 (cf. [17]). *Let* $\mathbf{C}$ *be a reactive system whose underlying* G*-category* $\mathbb{C}$ *has redex* G*RPOs. The largest bisimulation* $\sim$ *on* $\mathsf{GTS}(\mathbf{C})$ *is a congruence.*

The next three subsections complement this result by proving the expected corresponding theorems for trace and failure semantics, and by lifting them to the case of weak equivalences. Theorems and proofs in this section follow closely [11], as they are meant to show that GRPOs are as viable a tool as RPOs are.

2.1 Traces Preorder

Trace semantics [16] is a simple notion of equivalence which equates processes if they can engage in the same sequences of actions. Even though it lacks the fine discriminating power of branching time equivalences, viz. bisimulations, it is nevertheless interesting because many safety properties can be expressed as conditions on sets of traces.

We say that a sequence $f_1 \cdots f_n$ of labels of $\mathsf{GTS}(\mathbf{C})$ is a trace of a if

$$a \xrightarrow{f_1} \cdots \xrightarrow{f_n} a_{n+1}$$

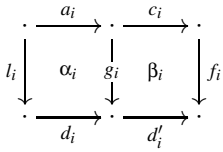
for some $a_1, \ldots, a_n$. The trace preorder $\lesssim_{\mathrm{tr}}$ is then defined as $a \lesssim_{\mathrm{tr}} b$ if all traces of a are also traces of b.

Theorem 2.3 (Trace Congruence). *$\lesssim_{\mathrm{tr}}$ is a congruence.*

Proof. Assume $a \lesssim_{\mathrm{tr}} b$. We prove that $ca \lesssim_{\mathrm{tr}} cb$ for all contexts $c \in \mathbb{C}$. Suppose that

$$ca = \bar{a}_1 \xrightarrow{f_1} \bar{a}_2 \cdots \bar{a}_n \xrightarrow{f_n} \bar{a}_{n+1}.$$

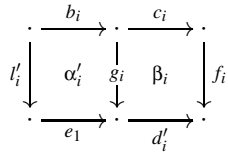
We first prove that there exist a sequence, for $i = 1, \ldots, n$,



where $a_1 = a,\ c_1 = c,\ c_{i+1} = d'_i,\ \bar{a}_i = c_i a_i,$ and each square is a GIPO.[1] The ith induction step proceeds as follows. Since $\bar{a}_i \xrightarrow{f_i} \bar{a}_{i+1}$, there exists $\gamma_i \colon f_i c_i a_i \Rightarrow \bar{d}_i l_i$, for some $\langle l_i, r_i \rangle \in \mathcal{R}$ and $\bar{d}_i \in \mathbb{D}$, with $\bar{a}_{i+1} = \bar{d}_i r_i$. Since $\mathbf{C}$ has redex GIPOs (cf. Definition 2.1), this can be split in two GIPOs: $\alpha_i \colon g_i a_i \Rightarrow d_i l_i$ and $\beta_i \colon f_i c_i \Rightarrow d'_i g_i$ (cf. diagram above). Take $a_{i+1} = d_i r_i$, and the induction hypothesis is maintained. In particular, we obtained a trace

$$a = a_1 \xrightarrow{g_1} a_2 \cdots a_n \xrightarrow{g_n} a_{n+1}$$

that, in force of the hypothesis $a \lesssim_{\mathrm{tr}} b$ must be matched by a corresponding trace of b. This means that, for $i = 1, .., n$, there exist GIPOs $\alpha'_i \colon g_i b_i \Rightarrow e_i l'_i$, for some $\langle l'_i, r'_i \rangle \in \mathcal{R}$ and $e_i \in \mathbb{D}$, once we take b_{i+1} to be $e_i r'_i$. We can then paste each of such GIPOs together with $\beta_i \colon f_i c_i \Rightarrow d'_i g_i$ obtained above and, using Lemma A.3, conclude that there exist GIPOs $f_i c_i b_i \Rightarrow d'_i e_i l'_i$, as in the diagram below.



which means $c_i b_i \xrightarrow{f_i} d'_i e_i r'_i$.

[1] Since the fact is not likely to cause confusion, we make no notational distinction between the arrows of $\mathbf{C}$ (e.g. in GRPOs diagrams) and the states and labels of $\mathsf{GTS}(\mathbf{C})$, where the latter are iso-classes of the former.

As $cb = c_1 b_1$, in order to construct a trace $cb = \bar{b}_1 \xrightarrow{f_1} \cdots \xrightarrow{f_n} \bar{b}_{n+1}$ and complete the proof, we only need to verify that for $i = 1, \ldots, n$, we have that $d'_i e_i r'_i = c_{i+1} b_{i+1}$. This follows at once, as $c_{i+1} = d'_i$ and $b_{i+1} = e_i r'_i$. □

2.2 Failures Preorder

Failure semantics [6] enhances trace semantics with limited branch-inspecting power. More precisely, failure sets allow the testing of when processes renounce the capability of engaging in certain actions.

Formally, for a a state of $\mathsf{GTS}(\mathbf{C})$, a *failure* of a is a pair $(f_1 \cdots f_n, X)$, where $f_1 \cdots f_n$ and X are respectively a sequence and a set of labels, such that:

- $f_1 \cdots f_n$ is a trace of a, $a \xrightarrow{f_1} \cdots \xrightarrow{f_n} a_{n+1}$;
- a_{n+1}, the final state of the trace, is *stable*, i.e. $a_{n+1} \not\longrightarrow$;
- a_{n+1} *refuses* X, i.e. $a_{n+1} \not\xrightarrow{x}$ for all $x \in X$.

The failure preorder $\lesssim_{\mathrm{f}}$ is defined as $a \lesssim_{\mathrm{f}} b$ if all failures of a are also failures of b. The proof of the following result can be found in [18].

Theorem 2.4 (Failures Congruence). *$\lesssim_{\mathrm{f}}$ is a congruence.*

2.3 Weak Equivalences

Theorems 2.2, 2.3, and 2.4 can be extended to weak equivalences, as outlined below.

For f a label of $\mathsf{GTS}(\mathbf{C})$ define a *weak transition* $a \overset{f}{\Longrightarrow} b$ to be a mixed sequence of transitions and reductions $a \longrightarrow\!\!\triangleright^* \xrightarrow{f} \longrightarrow\!\!\triangleright^* b$. Observe that this definition essentially identifies silent transitions in the LTS with reductions. As a consequence, care has to be taken to avoid interference with transitions $\xrightarrow{equi}$ synthesised from GRPOs and labelled by an equivalence. These transitions have essentially the same meaning as silent transitions (i.e. no context involved in the reduction), and must therefore be omitted in weak observations. This lead to consider the following definitions.

Definition 2.5 (Weak Traces and Failures). A sequence $f_1 \cdots f_n$ of *non-equivalence* labels of $\mathsf{GTS}(\mathbf{C})$ is a weak trace of a if

$$a \overset{f_1}{\Longrightarrow} a_1 \cdots a_{n-1} \overset{f_n}{\Longrightarrow} a_n$$

for some $a_1, \ldots, a_n$. The weak trace preorder is then defined accordingly.

A *weak failure* of a is a pair $(f_1 \cdots f_n, X)$, where $f_1 \cdots f_n$ and X are respectively a sequence and a set of *non-equivalence* labels, such that $f_1 \cdots f_n$ is a weak trace of a reaching a final state which is stable and refuses X. The weak trace preorder is defined accordingly.

Definition 2.6 (Weak Bisimulation). A symmetric relation $\mathcal{S}$ on $\mathsf{GTS}(\mathbf{C})$ is a weak bisimulation if for all $a \mathcal{S} b$

$$a \xrightarrow{f} a' \quad f \text{ not an equivalence}, \quad \text{implies } b \overset{f}{\Longrightarrow} b' \text{ with } a' \mathcal{S} b'$$
$$a \longrightarrow\!\!\triangleright a' \quad \text{implies } b \longrightarrow\!\!\triangleright^* b' \text{ with } a' \mathcal{S} b'$$

Using the definitions above Theorems 2.2, 2.3, and 2.4 can be lifted, respectively, to weak traces, failures and bisimulation.

It is worth remarking that the congruence results, however, only hold for contexts $c \in \mathbb{D}$, as it is well known that non reactive contexts (i.e. those c where $ca \longrightarrow\!\!\triangleright\, cb$ does not follow from $a \longrightarrow\!\!\triangleright\, b$, as e.g. the CSS context $c = c_0 + -$) do not preserve weak equivalences. Alternative definitions of weak bisimulations are investigated in [11], and they are applicable *mutatis mutandis* to GRPOs.

3 Bunches and Wires

The category of "bunches and wires" was introduced in [13] as a skeletal algebra of shared wirings, abstracting over the notion of *names* in, e.g., the π calculus. Although elementary, its structure is complex enough to lack RPOs.

A bunch context of type $m_0 \to m_1$ consists of an ordered set of m_1 trees of depth 1 containing exactly m_0 holes. Leaves are labelled from an alphabet $\mathcal{K}$.

Definition 3.1. The category of ***bunch contexts*** $\mathbf{Bun}_0$ has

- objects the finite ordinals, denoted $m_0, m_1, \ldots$
- arrows are bunch contexts $c = (X, \mathrm{char}, \mathrm{root}) : m_0 \to m_1$, where X is a finite carrier, $\mathrm{root} : m_0 + X \to m_1$ is a surjective function linking leaves (X) and holes (m_0) to their roots (m_1), and $\mathrm{char} : X \to \mathcal{K}$ is a leaf labelling function.

Composing $c_0 : m_0 \to m_1$ and $c_1 : m_1 \to m_2$ means filling the m_1 holes of c_1 with the m_1 trees of c_0. Formally, $c_1 c_0$ is $(X, \mathrm{root}, \mathrm{char})$ where

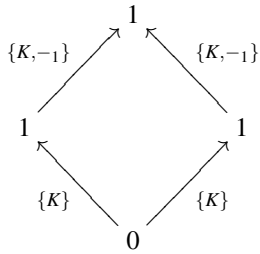
$$X = X_0 + X_1, \qquad \mathrm{root} = \mathrm{root}_1(\mathrm{root}_0 + \mathrm{id}_{X_1}), \qquad \mathrm{char} = [\mathrm{char}_0, \mathrm{char}_1],$$

where $+$ and $[\_,\_]$ are, resp., coproduct and copairing. Identities are $(\emptyset, !, \mathrm{id}) : m_0 \to m_0$.

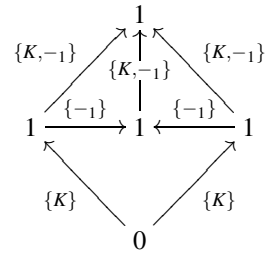
A ***homomorphism*** of bunch contexts $\rho : c \Rightarrow c' : m_0 \to m_1$ is a function $\rho : X \to X'$ which respects root and char, i.e. $\mathrm{root}'\rho = \mathrm{root}$ and $\mathrm{char}'\rho = \mathrm{char}$. An isomorphism is a bijective homomorphism. Isomorphic bunch contexts are equated, making composition associative and $\mathbf{Bun}_0$ a category.

A bunch context $c : m_0 \to m_1$ can be depicted as a string of m_1 nonempty multisets on $\mathcal{K} + m_0$, with the proviso that elements m_0 must appear exactly once in the string. In the examples, we represent elements of m_0 as numbered holes $-_i$.

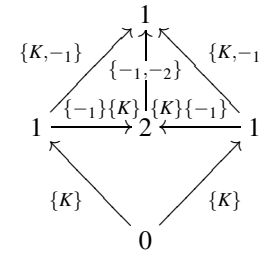
As we mentioned before, RPOs do not exist in $\mathbf{Bun}_0$. Indeed, consider *(i)* below together with the two candidates *(ii)* and *(iii)*. It is easy to show that these have no common "lower bound" candidate.



(*i*)



(*ii*)



(*iii*)

The point here is that by taking the arrows of $\mathbf{Bun}_0$ up to isomorphism we lose information about *how* bunch contexts equal each other. Diagram *(i)*, for instance, can be commutative in two different ways: the K in the bottom left part may corresponds either to the one in the bottom right or to the one in the top right, according to whether we read $\{K,-_1\}$ or $\{-_1,K\}$ for the top rightmost arrow. In order to track this information we endow $\mathbf{Bun}_0$ with its natural 2-categorical structure.

Definition 3.2. The 2-category of bunch contexts **Bun** has:

- objects the finite ordinals, denoted $m_0, m_1, \ldots$; we use **Ord** to denote the category of finite ordinals, and $\oplus$ for ordinal addition.
- arrows $c = (x, \mathrm{char}, \mathrm{root})\colon m_0 \to m_1$ consist of a finite ordinal x, a surjective function $\mathrm{root}\colon m_0 \oplus x \to m_1$ and a labelling function $\mathrm{char}\colon x \to \mathcal{K}$.
- 2-cells ρ are isomorphisms between bunches' carriers.

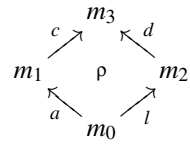
Composition of arrows and 2-cells is defined in the obvious way. Notice that since $\oplus$ is associative, composition in **Bun** is associative. Therefore **Bun** is a G-category.

Replacing the carrier set X with a finite ordinal x allows us to avoid the unnecessary burden of working in a bicategory, which would arise because sum on sets is only associative up to isomorphism. Observe that this simplification is harmless since the set theoretical identity of the elements of the carrier is irrelevant. We remark, however, that GRPOs are naturally a bicategorical notion and would pose no particular challenge in bicategories.

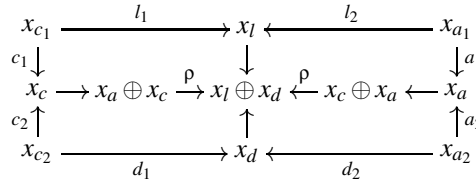
Theorem 3.3. Bun has GRPOs.

Proof. Here we give a basic account of the construction of a GRPO, but omit the proof of universality. In the following, we use only the fact that **Bun** is an extensive(cf. [2]) category with pushouts.

Suppose that we have



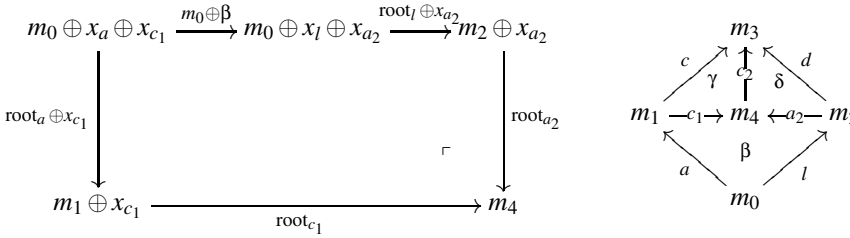
In the following diagram all the rectangles are pullbacks in **Ord** and all the outside arrows are coproduct injections.



Using the morphisms from the diagram above as building blocks, we can construct bijections $\gamma\colon x_c \to x_{c_1} \oplus x_{c_2}$, $\delta\colon x_{a_2} \oplus x_{c_2} \to x_d$ and $\beta\colon x_a \oplus x_{c_1} \to x_l \oplus x_{a_2}$ such that

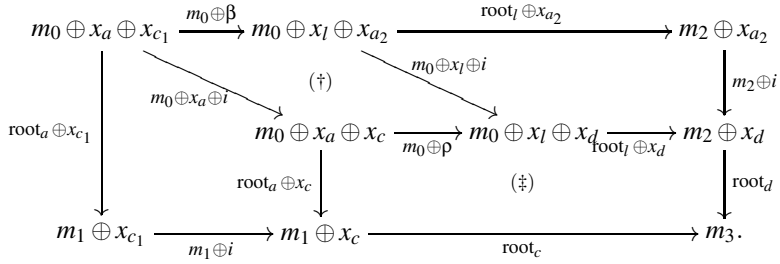
$$x_l \oplus \delta.\beta \oplus x_{c_2}.x_a \oplus \gamma = \rho. \tag{3}$$

Let root_{c_1} and root_{a_2} be the morphisms making *(i)* below



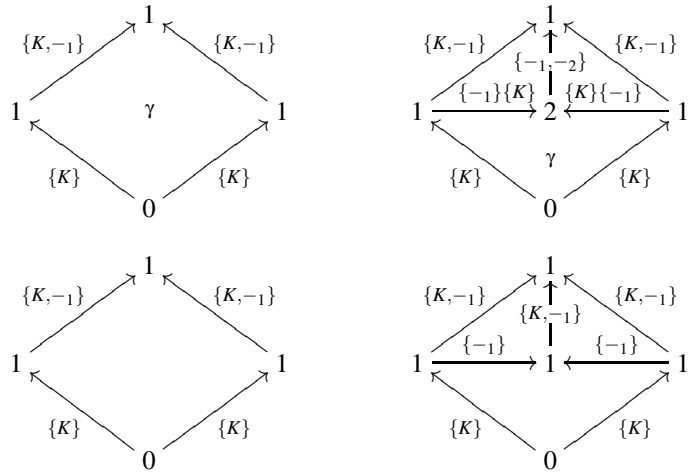
into a pushout diagram. We can define char_{c_1}, char_{a_2} and char_{c_2} in the obvious way.

Now consider the diagram below:



Region $(\dagger)$ can be verified to be commutative using (3) while region $(\ddagger)$ commutes since ρ is a homomorphism. Using the pushout property, we get a unique function $h\colon m_4 \to m_3$. Thus we define $\mathrm{root}_{c_2}\colon m_4 \oplus x_{c_2} \to m_3$ as $[h, \mathrm{root}_c\, i]$. It is easy to verify that this function is surjective. □

Example 3.4. Let $\gamma\colon 2 \to 2$ be the function taking $1 \mapsto 2$ and $2 \mapsto 1$. We give below on the right the GRPOs for the squares on the left.



4 2-Categories Vs Precategories

Other categories which, besides $\mathbf{Bun}_0$, lack RPOs include the closed *shallow action contexts* [11, 12] and *bigraph contexts* [15, 7]. The solution adopted by Leifer [12] and later by Milner [15] is to introduce a notion of a *well-supported precategory*, where the algebraic structures at hand are decorated by finite "support sets." The result is no longer a category – since composition of arrows is defined only if their supports are disjoint – but from any such precategory one can generate two categories which jointly allow the derivation of a bisimulation congruence via a *functorial reactive system*. These categories are the so-called *track* category, where support information is built into the objects, and the *support quotient* category, where arrows are quotiented by the support structure. The track category has enough RPOs and is mapped to the support quotient category via a well-behaved *functor*, so as to transport RPOs adequately.

In this section we present a translation from precategories to G-categories. The main result shows that the LTS derived using precategories and functorial reactive systems is identical to the LTS derived using GRPOs. We begin with a brief recapitulation of the definitions from [12].

Definition 4.1. A ***precategory*** $\mathbb{A}$ consists of the same data as a category. The composition operator $\circ$ is, however, a partial function which satisfies

1. for any arrow $f : A \to B$, $\mathrm{id}_B \circ f$ and $f \circ \mathrm{id}_A$ are defined and $\mathrm{id}_B \circ f = f = f \circ \mathrm{id}_A$;
2. for any $f : A \to B$, $g : B \to C$, $h : C \to D$, $(h \circ g) \circ f$ is defined iff $h \circ (g \circ f)$ is defined and then $(h \circ g) \circ f = h \circ (g \circ f)$.

Definition 4.2. Let Set_f be the category of finite sets. A ***well supported precategory*** is a pair $\langle \mathbb{A}, |-| \rangle$, where $\mathbb{A}$ is a precategory and $|-| : \mathrm{Arr}\,\mathbb{A} \to \mathsf{Set}_f$ is the so-called support function, satisfying:

1. $g \circ f$ is defined iff $|g| \cap |f| = \emptyset$, and if $g \circ f$ is defined then $|g \circ f| = |g| \cup |f|$;
2. $|\mathrm{id}_A| = \emptyset$.

For any $f : A \to B$ and any injective function ρ in Set_f the domain of which contains $|f|$ there exists an arrow $\rho \cdot f : A \to B$ called the ***support translation*** of f by ρ. The following axioms are to be satisfied.

1. $\rho \cdot \mathrm{id}_A = \mathrm{id}_A$;
2. $\mathrm{id}_{|f|} \cdot f = f$;
3. $\rho_0|f| = \rho_1|f|$ implies $\rho_0 \cdot f = \rho_1 \cdot f$;
4. $\rho \cdot (g \circ f) = \rho \cdot g \circ \rho \cdot f$;
5. $(\rho_1 \circ \rho_0) \cdot f = \rho_1 \cdot (\rho_0 \cdot f)$;
6. $|\rho \cdot f| = \rho|f|$.

We illustrate these definitions giving a precategorical definition of bunches and wiring (viz. §3).

Example 4.3 (Bunches). The precategory of bunch contexts **A-Bun** has objects and arrows as in $\mathbf{Bun}_0$. However, differently from $\mathbf{Bun}_0$, they are not taken up to isomorphism here. The support of $c = (X, \mathrm{char}, \mathrm{root})$ is X. Composition $c_1 c_0 = (X, \mathrm{char}, \mathrm{root}) : m_0 \to m_2$ of $c_0 : m_0 \to m_1$ and $c_1 : m_1 \to m_2$ is defined if $X_0 \cap X_1 = \emptyset$ and, if so, we have $X = X_0 \cup X_1$. Functions char and root are defined in the obvious way. The identity

arrows are the same as in $\mathbf{Bun_0}$. Given an injective function $\rho\colon X \to Y$, the support translation $\rho \cdot c$ is $(\rho X, \mathrm{char}\,\rho^{-1}, \mathrm{root}\,(\mathrm{id}_{m_0} + \rho^{-1}))$. It is easy to verify that this satisfies the axioms of precategories.

The definitions below recall the construction of the track and the support quotient categories from a well-supported precategory.

Definition 4.4. The ***track*** of $\mathbb{A}$ is a category $\widehat{\mathbb{C}}$ with

- objects: pairs $\langle A, M\rangle$ where $A \in \mathbb{A}$ and $M \in \mathsf{Set}_f$;
- arrows: $\langle A, M\rangle \xrightarrow{f} \langle B, N\rangle$ where $f\colon A \to B$ is in $\mathbb{A}$, $M \subseteq N$ and $|f| = N \backslash M$.

Composition of arrows is as in $\mathbb{A}$. Observe that the definition of $|f|$ ensures that composition is total. We leave it to the reader to check that this defines a category (cf. [12]).

Definition 4.5. The ***support quotient*** of $\mathbb{A}$ is a category $\mathbb{C}$ with

- objects: as in $\mathbb{A}$;
- arrows: equivalence classes of arrows of $\mathbb{A}$, where f and g are equated if there exist a bijective ρ such that $\rho \cdot f = g$.

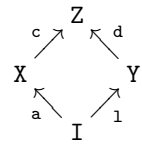
The support quotient is the category of interest, and it is the underlying category of the reactive system under scrutiny.

Example 4.6 (Bunches). The support quotient of **A-Bun** is $\mathbf{Bun}_0$.

There is an obvious functor $F\colon \widehat{\mathbb{C}} \to \mathbb{C}$, the support-quotienting functor. Henceforward we suppose that the precategory $\mathbb{A}$ has a distinguished object I. In the following we use the typewriter font for objects and arrows of $\widehat{\mathbb{C}}$. We make the notational convention that any $\mathtt{A}$ and $\mathtt{f}$ in $\widehat{\mathbb{C}}$ are such that $F(\mathtt{A}) = A$ and $F(\mathtt{f}) = f$.

Definition 4.7 (The LTS). The LTS $\mathbf{FLTS}^c(\mathbb{C})$ has

- States: arrows $a\colon 0 \to n$ in $\mathbb{C}$;
- Transitions: $a \xrightarrow{c} dr$ if and only if there exist $\mathtt{a}, \mathtt{l}, \mathtt{c}, \mathtt{d}$ in $\hat{\mathbb{C}}$ with $\langle F(\mathtt{l}), r\rangle \in \mathcal{R}$, $F(\mathtt{d}) \in \mathbb{D}$, and such that



is an IPO.

It is proved in [12] that the support-quotienting functor F satisfies the properties required for the theory of functorial reactive systems [11, 12]. Thus, for instance, if the category $\widehat{\mathbb{C}}$ has enough RPOs, then the bisimulation on $\mathbf{FLTS}^c(\mathbb{C})$ is a congruence.

All the theory presented so far can be elegantly assimilated into the theory of GRPOs. In [12], Leifer predicted instead of precategories, one could consider a bicategorical notion of RPO in a bicategory of supports. This is indeed the case, with GRPOs

being the bicategorical notion of RPO. However, working with ordinals for support sets we can avoid the extra complications bicategories as in the case of **Bun**. It is worth noticing, however, that a bicategory of supports as above and the G-category define below would be biequivalent (cf. [20]).

In the following, we make use of a chosen isomorphism $t_x : x \to \mathrm{ord}(x)$, where for any finite set x, $\mathrm{ord}(x)$ denotes the finite ordinal of the same cardinality. There is an equivalence of categories $F : \mathbf{Set}_f \to \mathbf{Ord}$ which sends x to $\mathrm{ord}(x)$ and, on morphisms, $f : x \to y$ to $t_y f t_x^{-1} : \mathrm{ord}(x) \to \mathrm{ord}(y)$.

Definition 4.8 (G-category of Supports). Given a well-supported precategory $\mathbb{A}$, the G-category of supports $\mathbb{B}$ has

- objects: as in $\mathbb{A}$;
- arrows: $f : A \to B$ where $f : A \to B$ is an arrow of $\mathbb{A}$ and $|f|$ is an ordinal;
- 2-cells: $\rho : f \Rightarrow g$, where ρ is a "structure preserving" support bijection, that is $\rho \cdot f = g$ in $\mathbb{A}$.

Composition is defined as follows. Given $f : A \to B$ and $g : B \to C$,

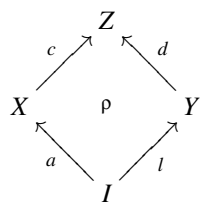
$$g \circ_{\mathbb{B}} f = i_2 \cdot g \circ_{\mathbb{A}} i_1 \cdot f$$

where $|f| \xrightarrow{i_1} |f| \oplus |g| \xleftarrow{i_2} |g|$ is the chosen coproduct diagram in **Ord**.

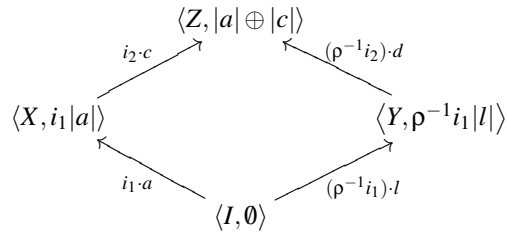
The following theorem guarantees that the LTS generated is the same as the one generated with the more involved theory of functorial reactive systems.

Theorem 4.9. $\mathbf{FLTS}^c(\mathbb{A}) = \mathbf{GTS}(\mathbb{B})$.

Proof. It is enough to present translations between GIPOs in $\mathbb{B}$ and IPOs in $\widehat{\mathbb{C}}$ which preserve the resulting label in the derived LTS. We present the translations, but omit the straightforward proofs. Suppose that *(i)* below

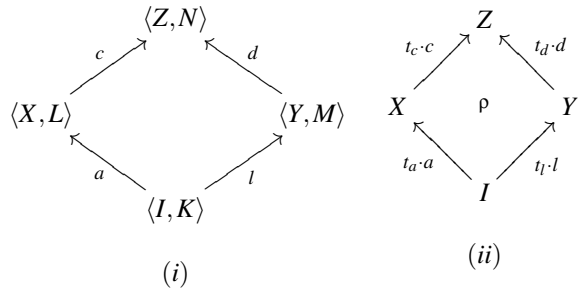


(i)



(ii)

is a GIPO in $\mathbb{B}$. Then we claim that *(ii)* is an IPO in $\mathbb{C}$. Note that *(ii)* is commutative since ρ is a structure-preserving support bijection. Going the other way, suppose that *(i)* below



is an IPO in $\mathbb{C}$. Then *(ii)* is a GIPO in $\mathbb{B}$ where ρ is

$$|t_a \cdot a| \oplus |t_c \cdot c| \xrightarrow{t_a^{-1} \oplus t_c^{-1}} |a| \cup |c| = |l| \cup |d| \xrightarrow{t_l \cup t_d} |t_l \cdot l| \oplus |t_d \cdot d|.$$

□

Example 4.10 (Bunches). The 2-category of supports of the precategory **A-Bun** is **Bun**. Note that a "structure preserving" support bijection is a bunch homomorphism. Indeed, $\rho\colon (X, \mathrm{char}, \mathrm{root}) \to (X', \mathrm{char}', \mathrm{root}')$ if $X' = \rho X$, $\mathrm{char}' = \mathrm{char}\,\rho^{-1}$ and $\mathrm{root}' = \mathrm{root}(\mathrm{id} \oplus \rho^{-1})$ which is the same as saying $\mathrm{char} = \mathrm{char}'\,\rho$ and $\mathrm{root} = \mathrm{root}'(\mathrm{id} \oplus \rho)$.

5 Conclusion

We have extended our theory of GRPOs initiated in previous work in order to strengthen existing techniques for deriving operational congruences for reduction systems in the presence of non trivial structural congruences. In particular, this paper has shown that previous theories can be recast using G-reactive systems and GRPOs at no substantial additional complexity. Also, we proved that the theory is powerful enough to handle the examples considered so far in the literature. Therefore, we believe that it constitutes a natural starting point for future investigations towards a fully comprehensive theory.

It follows from Theorem 4.9 that G-categories are at least as expressive as well-supported precategories. A natural consideration is whether a reverse translation may exist. We believe that this is not the case, as general G-categories appear to carry more information than precategories.

References

[1] M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication interference in mobile boxed ambients. In *Foundations of Software Technology and Theoretical Computer Science, FST&TCS 02*, volume 2556 of *Lecture Notes in Computer Science*, pages 71–84. Springer, 2002.

[2] A. Carboni, S. Lack, and R. F. C. Walters. Introduction to extensive and distributive categories. *Journal of Pure and Applied Algebra*, 84(2):145–158, February 1993.

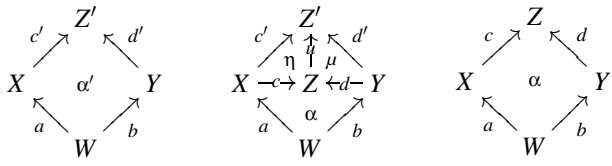
[3] G. Castagna and F. Zappa Nardelli. The seal calculus revised. In *Foundations of Software Technology and Theoretical Computer Science, FST&TCS 02*, volume 2556 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2002.

[4] J. C. Godskesen, T. Hildebrandt, and V. Sassone. A calculus of mobile resources. In *Int. Conf. on Concurrency Theory, CONCUR 02*, volume 2421 of *Lecture Notes in Computer Science*, pages 272–287. Springer, 2002.
[5] M. Hennessy and M. Merro. Bisimulation congruences in safe ambients. In *Principles of Programming Languages, POPL 02*, pages 71–80. ACM Press, 2002.
[6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
[7] O. H. Jensen and R. Milner. Bigraphs and transitions. In *Principles of Programming Languages, POPL 03*. ACM Press, 2003.
[8] G. M. Kelly. Elementary observations on 2-categorical limits. *Bull. Austral. Math. Soc.*, 39:301–317, 1989.
[9] G. M. Kelly and R. H. Street. Review of the elements of 2-categories. *Lecture Notes in Mathematics*, 420:75–103, 1974.
[10] F. W. Lawvere. Functorial semantics of algebraic theories. *Proceedings, National Academy of Sciences*, 50:869–873, 1963.
[11] J. Leifer. *Operational congruences for reactive systems*. Phd thesis, University of Cambridge, 2001.
[12] J. Leifer. Synthesising labelled transitions and operational congruences in reactive systems, parts 1 and 2. Technical Report RR-4394 and RR-4395, INRIA Rocquencourt, 2002.
[13] J. Leifer and R. Milner. Deriving bisimulation congruences for reactive systems. In *Int. Conf. on Concurrency Theory, CONCUR 00*, Lecture Notes in Computer Science, pages 243–258. Springer, 2000.
[14] R. Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996.
[15] R. Milner. Bigraphical reactive systems: Basic theory. Technical Report 523, Computer Laboratory, University of Cambridge, 2001.
[16] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
[17] V. Sassone and P. Sobociński. Deriving bisimulation congruences: A 2-categorical approach. *Electronic Notes in Theoretical Computer Science*, 68(2), 2002.
[18] V. Sassone and P. Sobociński. Deriving bisimulation congruences: 2-categories vs. precategories. Technical Report RS-03-1, BRICS, January 2003.
[19] P. Sewell. From rewrite rules to bisimulation congruences. *Lecture Notes in Computer Science*, 1466:269–284, 1998.
[20] R. H. Street. Fibrations in bicategories. *Cahiers de topologie et géométrie différentielle*, XXI-2:111–159, 1980.

A Basic Properties of GRPOs

The next two lemmas explain the relationships between GRPOs and GIPOs.

Lemma A.1 (GIPOs from GRPOs). *If $\langle Z, c, d, u, \alpha, \eta, \mu \rangle$ is a GRPO for (i) below, as illustrated in (ii), then (iii) is a GIPO.*

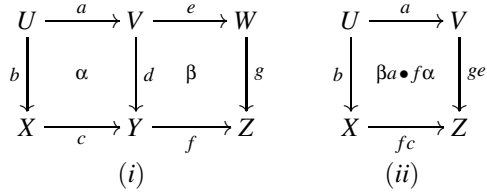


(*i*) (*ii*) (*iii*)

Lemma A.2 (GRPOs from GIPOs). *If square (iii) above is a* GIPO*, (i) has a* GRPO*, and* $\langle Z, c, d, u, \alpha, \eta, \mu \rangle$ *is a candidate for it as shown in (ii), then* $\langle Z, c, d, u, \alpha, \eta, \mu \rangle$ *is a* GRPO *for (i).*

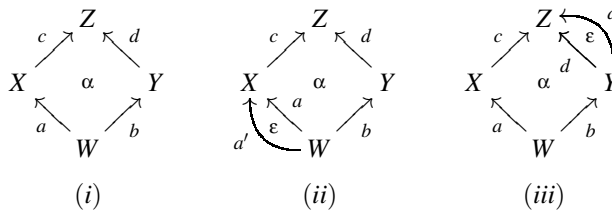
The following lemmas from [17] state the basic properties of GRPOs.

Lemma A.3. Suppose that diagram *(ii)* below has a GRPO.



1. If both squares in *(i)* are GIPOs then the rectangle of *(i)* is a GIPO
2. If the left square and the rectangle of *(i)* are GIPOs then so is the right square.

Lemma A.4. Suppose that diagram *(i)* below is a GIPO.



Then the regions obtained by pasting the 2-cells in *(ii)* and *(iii)* are GIPOs.

On the Structure of Inductive Reasoning: Circular and Tree-Shaped Proofs in the μ-Calculus

Christoph Sprenger[1]* and Mads Dam[2]**

[1] INRIA Sophia Antipolis, France
(sprenger@sophia.inria.fr)
[2] Royal Institute of Technology, Kista, Sweden
(mfd@imit.kth.se)

Abstract. In this paper we study induction in the context of the first-order μ-calculus with explicit approximations. We present and compare two Gentzen-style proof systems each using a different type of induction. The first is based on finite proof trees and uses a local well-founded induction rule, while the second is based on (finitely represented) ω-regular proof trees and uses a global induction discharge condition to ensure externally that all inductive reasoning is well-founded. We give effective procedures for the translation of proofs between the two systems, thus establishing their equivalence.

1 Introduction

Induction is the fundamental finitary method available in mathematics and computer science to generate and reason about finite and infinite objects. Three main proof-theoretic approaches to explicit induction[1] can be identified:

1. well-founded induction,
2. Scott/Park fixed point induction (cf. [7]), and
3. cyclic proofs, based on the idea of reducing induction to a global well-foundedness check on a locally sound inference graph.

As general approaches to inductive reasoning the former two are clearly the dominant ones. However, the value of cyclic reasoning in the decidable case has been demonstrated quite unequivocally by now. Examples are the well-established equivalence between monadic second-order logic, temporal logics (including various μ-calculi) and automata on infinite objects [14] as well as the usefulness of

* Research done mainly while at Swedish Institute of Computer Science (SICS), supported by Swiss European Fellowship 83EU-065536, and completed at INRIA, supported by an ERCIM fellowship.
** Supported by the Swedish Research Council grant 621-2001-2637, "Semantics and Proof of Programming Languages"
[1] As opposed to implicit induction (cf. [3]), based on Knuth-Bendix completion.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 425–440, 2003.

automata for obtaining decidability results and efficient algorithms [15]. Similar claims can be made concerning the usefulness of tableau-based techniques for obtaining completeness results in modal and temporal logic [6]. In the context of theorem proving and (undecidable) proof search, however, cyclic reasoning has received little attention. In our opinion, this situation deserves to be remedied. Our claim is that it facilitates proof search, mainly due to its ability to delay decisions concerning induction strategy as far as possible. Although it is too early for any real conclusions on the validity of this claim, the experiences with cyclic proofs for the μ-calculus using the EVT theorem prover [1, 5] are encouraging.

In this paper, we address the fundamental question of the relative deductive power of cyclic proofs versus well-founded induction, the latter being the yardstick by which other formalisations must be compared. Our investigation is based on Park's first-order μ-calculus [8], which provides a minimal setting to study formalised induction. In this context, cyclic reasoning underpins work on model checking [13], completeness of the modal μ-calculus [16], and, more recently, Gentzen-type proof systems for compositional verification [4, 11, 10]. We establish effective translations between two Gentzen-style proof systems: one, $\mathcal{S}_{loc}$, for well-founded (local) induction, and the other, $\mathcal{S}_{glob}$, based on (finitely represented) ω-regular proof trees using an external global induction discharge condition ensuring the well-foundedness. We work in an extension of the basic μ-calculus with explicit approximations [4] and ordinal quantification [10] (Sect. 2). Inductive reasoning in both proof systems rests on this extension. In system $\mathcal{S}_{loc}$ (Sect. 3), it is supported by a single local induction rule, an instantiation of the the well-known rule of well-founded induction to ordinals. In system $\mathcal{S}_{glob}$ (Sect. 4), the global induction discharge condition organises the basic cycles, called *repeats*, into a partial *induction order,* assigns a progressing induction variable to each repeat and requires each repeat to preserve (i.e. not increase) the variables of repeats above it in the induction order. This condition ensures that the induction associated with each strongly connected subgraph is well-founded. For the translation from $\mathcal{S}_{loc}$ to $\mathcal{S}_{glob}$ (Sect. 5) it is sufficient to show that the local induction rule of $\mathcal{S}_{loc}$ is derivable in $\mathcal{S}_{glob}$. The translation in the other direction (Sect. 6) is more involved and generally proceeds in two stages. We first present a direct translation for $\mathcal{S}_{glob}$-proofs, where the inductive structure represented in the induction order matches the proof tree structure. Then, we discuss an exponential time algorithm, which unfolds arbitrary cyclic proofs until they are in the form required by the direct translation.

We think that, by clearly exhibiting the underlying structures and their relationships, our present formal treatment sheds some new light on the various approaches to inductive reasoning. An important benefit from using explicit approximations is that it largely decouples our constructions from the actual language (here, the μ-calculus), thus strongly suggesting that they can support lazy-style global induction in other contexts such as type theories with inductive definitions [9]. Barthe et al. [2], for instance, points in this direction by proposing a type system ensuring termination of recursive functions based on approximations of inductive types. Finally, an interesting practical implication of our result

is that (assuming size blow-ups can be prevented) it permits standard theorem provers to be gracefully interfaced with the μ-calculus based EVT prover.

Set-theoretic preliminaries Let $G = (A, R \subseteq A \times A)$ be a graph. We say that G is a *forest* if $(a,b) \in R$ and $(a,c) \in R$ imply $b = c$. A *tree* is a forest with a unique *root node* $r \in A$ such that there is no $a \in A$ with $(a,r) \in R$. We call G *forest-like* if $(a,b) \in R$ and $(a,c) \in R$ imply $b = c$ or $(b,c) \in R \cup R^{-1}$. A subset $C \subseteq A$ is *strongly R-connected* if for any two $x, y \in C$ we have that $(x,y) \in R^*$. C is *weakly R-connected* if $R \cup R^{-1}$ is strongly R-connected. We sometimes call $C \subseteq A$ a subgraph of G and mean the subgraph $(C, R \cap C \times C)$ induced by C. A strongly connected subgraph (SCS) $C \subseteq A$ is *non-trivial* if $R \cap C \times C \neq \varnothing$.

2 μ-Calculus with Explicit Approximations

Let $\mathbf{2} = \{0,1\}$ be the two-point lattice and let $\mathsf{Pred}(S) = \mathbf{2}^S$ be the lattice of predicates over S ordered pointwise. For a monotone map $f : \mathsf{Pred}(S) \to \mathsf{Pred}(S)$ we define the *ordinal approximation* $\mu^\alpha f$ and the *fixed point* μf by

$$\begin{array}{ll} \mu^0 f = \lambda x.0 & \mu^\gamma f = \bigvee_{\alpha<\gamma} \mu^\alpha f \quad \text{for limit ordinals } \gamma \\ \mu^{\alpha+1} f = f(\mu^\alpha f) & \mu f = \bigvee_a \mu^\alpha f \end{array}$$

Proposition 1. *Let $f : \mathsf{Pred}(S) \to \mathsf{Pred}(S)$ be a monotone map. Then*

1. *μf is the least fixed point of f (Knaster-Tarski), and*
2. *$\mu^\alpha f = \bigvee_{\beta<\alpha} f(\mu^\beta f)$.*

We assume countably infinite sets of individual variables $x, y, z \ldots \in V_I$, of predicate variables $X, Y, Z, \ldots \in V_P$ of each arity $n \geq 0$, and of ordinal variables $\iota, \kappa, \lambda, \ldots \in V_O$. Let t range over the terms of some signature Σ.

Definition 1. (Syntax) *The syntax of* μ-calculus formulas ϕ and predicates Φ over Σ *is inductively defined by*

$$\begin{array}{l} \phi ::= t = t' \mid \kappa' < \kappa \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \exists x.\phi \mid \exists \kappa.\phi \mid \Phi(\overline{t}) \\ \Phi ::= X \mid \mu X(\overline{x}).\phi \mid \mu^\kappa X(\overline{x}).\phi \end{array}$$

with the restriction that the arities of Φ and $\overline{t}$ match in $\Phi(\overline{t})$ and the formation of both $\mu X(\overline{x}).\phi$ and $\mu^\kappa X(\overline{x}).\phi$ is subject to the conditions that (i) the arities of X and $\overline{x}$ match, and (ii) all occurrences of X in ϕ appear under an even number of negations (formal monotonicity).

Zero-ary predicates are identified with formulas. We call formulas of the form $\kappa' < \kappa$ *ordinal constraints*. The sets of free and bound variables of formulas and predicates are defined as expected, in particular, $\mathrm{fv}(\mu X(\overline{x}).\phi) = \mathrm{fv}(\phi) - \{X, \overline{x}\}$ and $\mathrm{fv}(\mu X^\kappa(\overline{x}).\phi) = (\mathrm{fv}(\phi) - \{X, \overline{x}\}) \cup \{\kappa\}$. We will identify formulas that are equal up to renaming of bound variables. Dual connectives are defined in the usual way, with $\nu X(\overline{x}).\phi = \neg\mu X(\overline{x}).\neg\phi[\neg X/X]$ and the approximated

fixed point similarly. We also use bounded ordinal quantifiers defined by the abbreviations $\exists \iota{<}\kappa.\phi = \exists \iota.\, \iota{<}\kappa \wedge \phi$ and $\exists \iota{\leq}\kappa.\phi = (\exists \iota{<}\kappa.\phi) \vee \phi[\kappa/\iota]$.

Given a Σ-model $\mathcal{M} = (\mathcal{A}, \rho)$ ($\mathcal{A}$ is the structure and ρ the interpretation) the semantics interprets a μ-calculus formula ϕ as an element $\|\phi\|_{\mathcal{M}} \in \mathbf{2}$ and an n-ary predicate Φ as an element $\|\Phi\|_{\mathcal{M}} \in \mathsf{Pred}(|A|^n)$. We often drop $\mathcal{M}$ and write $\|\phi\|_\rho$ and $\|\Phi\|_\rho$ if the structure $\mathcal{A}$ is clear from the context.

Definition 2. (Semantics) *Given a signature Σ and a Σ-model $(\mathcal{A}, \rho)$ define the semantics of μ-calculus formulas ϕ and predicates Φ over Σ inductively by*

$$
\begin{array}{lll}
\|t = t'\|_\rho & = \textit{if } \|t\|_\rho = \|t'\|_\rho \textit{ then } 1 \textit{ else } 0 & \|\exists \kappa.\phi\|_\rho = \bigvee_\beta \|\phi\|_{\rho[\beta/\kappa]} \\
\|\kappa' {<} \kappa\|_\rho & = \textit{if } \rho(\kappa') < \rho(\kappa) \textit{ then } 1 \textit{ else } 0 & \|\Phi(\overline{t})\|_\rho = \|\Phi\|_\rho(\|\overline{t}\|_\rho) \\
\|\neg\phi\|_\rho & = 1 - \|\phi\|_\rho & \|X\|_\rho = \rho(X) \\
\|\phi_1 \wedge \phi_2\|_\rho & = \min\{\|\phi_1\|_\rho, \|\phi_2\|_\rho\} & \|\mu X(\overline{x}).\phi\|_\rho = \mu\Psi \\
\|\exists x.\phi\|_\rho & = \bigvee_{a \in |A|} \|\phi\|_{\rho[a/x]} & \|\mu^\kappa X(\overline{x}).\phi\|_\rho = \mu^{\rho(\kappa)}\Psi
\end{array}
$$

where $\|\overline{t}\|_\rho$ is defined as usual and $\Psi = \lambda P.\lambda \overline{a}.\|\phi\|_{\rho[P/X, \overline{a}/\overline{x}]}$ in the clauses for fixed point and approximation predicates.

Given a model $\mathcal{M} = (\mathcal{A}, \rho)$, we extend the valuation ρ a posteriori to terms t and formulas ϕ by defining $\rho(t) = \|t\|_\rho$ and $\rho(\phi) = \|\phi\|_\rho$. This allows us to compose substitutions θ with environments ρ as in $\rho \circ \theta$. We say that a model $\mathcal{M} = (\mathcal{A}, \rho)$ *satisfies* a formula ϕ, written $\mathcal{M} \models \phi$, if $\|\phi\|_\rho = 1$. A formula ϕ is *valid* if $\mathcal{M} \models \phi$ for all models $\mathcal{M}$.

3 Local Induction: The System $\mathcal{S}_{loc}$

In this section we introduce the Gentzen-type proof system $\mathcal{S}_{loc}$ for local well-founded induction. It shares most definitions and proof rules with the system $\mathcal{S}_{glob}$ for global induction presented in the next section.

Sequents The *sequents* of both proof systems are of the form $\Gamma \vdash_{\mathcal{O}} \Delta$, where Γ and Δ are finite multisets of formulas and $\mathcal{O}$ is a finite set of ordinal variables. A sequent is *well-formed* if all ordinal variables occurring free in Γ or Δ are elements of $\mathcal{O}$. We tacitly restrict our attention to well-formed sequents. The set of free variables of a sequent is defined by $\mathrm{fv}(\Gamma \vdash_{\mathcal{O}} \Delta) = \mathrm{fv}(\Gamma \cup \Delta) \cup \mathcal{O}$. Substitutions are extended to multisets of formulas by defining $\Gamma[\theta] = \{\phi[\theta] \mid \phi \in \Gamma\}$. In sequents we often write $\mathcal{O}, \kappa$ for $\mathcal{O} \cup \{\kappa\}$.

Given a Σ-model $\mathcal{M} = (\mathcal{A}, \rho)$ we say that $\mathcal{M}$ *satisfies* a sequent $\Gamma \vdash_{\mathcal{O}} \Delta$ if $\mathcal{M} \models \phi$ for all $\phi \in \Gamma$ implies that $\mathcal{M} \models \psi$ for some $\psi \in \Delta$. A model $\mathcal{M}$ *falsifies* a sequent $\Gamma \vdash_{\mathcal{O}} \Delta$ if $\mathcal{M}$ does not satisfy $\Gamma \vdash_{\mathcal{O}} \Delta$. The sequent $\Gamma \vdash_{\mathcal{O}} \Delta$ is *valid* if it is satisfied in all models.

Structural Rules

$$\text{(Id)}\ \frac{}{\Gamma, \phi \vdash_{\mathcal{O}} \phi, \Delta} \qquad \text{(Weak)}\ \frac{\Gamma \vdash_{\mathcal{O}} \Delta}{\Gamma' \vdash_{\mathcal{O}'} \Delta'}\ \begin{array}{l}\Gamma' \subseteq \Gamma \\ \Delta' \subseteq \Delta \\ \mathcal{O}' \subseteq \mathcal{O}\end{array}$$

$$\text{(Cut)}\ \frac{\Gamma \vdash_{\mathcal{O}} \Delta}{\Gamma, \phi \vdash_{\mathcal{O}} \Delta \quad \Gamma \vdash_{\mathcal{O}} \phi, \Delta} \qquad \text{(Subst)}\ \frac{\Gamma[\theta] \vdash_{\mathcal{O}[\theta]} \Delta[\theta]}{\Gamma \vdash_{\mathcal{O}} \Delta}$$

Logical and Equality Rules

$$\text{(}\neg\text{-L)}\ \frac{\Gamma, \neg\phi \vdash_{\mathcal{O}} \Delta}{\Gamma \vdash_{\mathcal{O}} \phi, \Delta} \qquad \text{(}\neg\text{-R)}\ \frac{\Gamma \vdash_{\mathcal{O}} \neg\phi, \Delta}{\Gamma, \phi \vdash_{\mathcal{O}} \Delta}$$

$$\text{(}\wedge\text{-L)}\ \frac{\Gamma, \phi_1 \wedge \phi_2 \vdash_{\mathcal{O}} \Delta}{\Gamma, \phi_1, \phi_2 \vdash_{\mathcal{O}} \Delta} \qquad \text{(}\wedge\text{-R)}\ \frac{\Gamma \vdash_{\mathcal{O}} \phi_1 \wedge \phi_2, \Delta}{\Gamma \vdash_{\mathcal{O}} \phi_1, \Delta \quad \Gamma \vdash_{\mathcal{O}} \phi_2, \Delta}$$

$$\text{(}\exists\text{-L)}\ \frac{\Gamma, \exists x.\phi \vdash_{\mathcal{O}} \Delta}{\Gamma, \phi \vdash_{\mathcal{O}} \Delta}\ x \notin \mathrm{fv}(\Gamma, \Delta) \qquad \text{(}\exists\text{-R)}\ \frac{\Gamma \vdash_{\mathcal{O}} \exists x.\phi, \Delta}{\Gamma \vdash_{\mathcal{O}} \phi[t/x], \Delta}$$

$$\text{(}=\text{-L)}\ \frac{\Gamma[t_2/x], t_1 = t_2 \vdash_{\mathcal{O}} \Delta[t_2/x]}{\Gamma[t_1/x] \vdash_{\mathcal{O}} \Delta[t_1/x]} \qquad \text{(}=\text{-R)}\ \frac{\Gamma \vdash_{\mathcal{O}} t = t, \Delta}{}$$

Fixed Point Rules

$$\text{(}\mu_1\text{-L)}\ \frac{\Gamma, (\mu X(\overline{x}).\phi)(\overline{t}) \vdash_{\mathcal{O}} \Delta}{\Gamma, \exists\kappa.(\mu^\kappa X(\overline{x}).\phi)(\overline{t}) \vdash_{\mathcal{O}} \Delta} \qquad \text{(}\mu_0\text{-R)}\ \frac{\Gamma \vdash_{\mathcal{O}} (\mu X(\overline{x}).\phi)(\overline{t}), \Delta}{\Gamma \vdash_{\mathcal{O}} \phi[\mu X(\overline{x}).\phi/X, \overline{t}/\overline{x}], \Delta}$$

$$\text{(}\mu^\kappa\text{-L)}\ \frac{\Gamma, (\mu^\kappa X(\overline{x}).\phi)(\overline{t}) \vdash_{\mathcal{O}} \Delta}{\Gamma, \exists\kappa' < \kappa.\phi[\mu^{\kappa'} X(\overline{x}).\phi/X, \overline{t}/\overline{x}] \vdash_{\mathcal{O}} \Delta}$$

$$\text{(}\mu^\kappa\text{-R)}\ \frac{\Gamma \vdash_{\mathcal{O}} (\mu^\kappa X(\overline{x}).\phi)(\overline{t}), \Delta}{\Gamma \vdash_{\mathcal{O}} \exists\kappa' < \kappa.\phi[\mu^{\kappa'} X(\overline{x}).\phi/X, \overline{t}/\overline{x}], \Delta}$$

Ordinal Rules

$$\text{(}\exists\kappa\text{-L)}\ \frac{\Gamma, \exists\kappa.\phi \vdash_{\mathcal{O}} \Delta}{\Gamma, \phi \vdash_{\mathcal{O},\kappa} \Delta}\ \kappa \notin \mathcal{O} \qquad \text{(}\exists\kappa\text{-R)}\ \frac{\Gamma \vdash_{\mathcal{O}} \exists\kappa.\phi, \Delta}{\Gamma \vdash_{\mathcal{O}} \phi[\iota/\kappa], \Delta}\ \iota \in \mathcal{O}$$

$$\text{(}<\text{-L)}\ \frac{\Gamma, \kappa < \kappa \vdash_{\mathcal{O}} \Delta}{} \qquad \text{(}<\text{-R)}\ \frac{\Gamma \vdash_{\mathcal{O}} \kappa' < \kappa, \Delta}{\Gamma \vdash_{\mathcal{O}} \kappa' < \kappa'', \Delta \quad \Gamma \vdash_{\mathcal{O}} \kappa'' < \kappa, \Delta}$$

Table 1. The proof rules shared by $\mathcal{S}_{loc}$ and $\mathcal{S}_{glob}$

$$\text{(Ind-L)}\ \frac{\Gamma, \exists\kappa.\phi \vdash_{\mathcal{O}} \Delta}{\Gamma, \phi \vdash_{\mathcal{O},\kappa} \exists\kappa' < \kappa.\phi[\kappa'/\kappa], \Delta}\ \kappa \notin \mathcal{O}$$

$$\text{(Ind-R)}\ \frac{\Gamma \vdash_{\mathcal{O}} \forall\kappa.\phi, \Delta}{\Gamma, \forall\kappa' < \kappa.\phi[\kappa'/\kappa] \vdash_{\mathcal{O},\kappa} \phi, \Delta}\ \kappa \notin \mathcal{O}$$

Table 2. The local induction rules of $\mathcal{S}_{loc}$

Proof System The proof system is presented in two parts. Table 1 shows the basic set of proof rules, common to both $\mathcal{S}_{loc}$ and $\mathcal{S}_{glob}$. They are presented in tableau-style with the conclusion above the line and the premises below. Fixed point rule (μ_1-L) is the essential device which introduces ordinal approximations. Note the asymmetry of rules (μ_1-L) and (μ_0-R). However, one can show that the symmetric rules (μ_0-L) and (μ_1-R) are derivable in $\mathcal{S}_{loc}$. The local proof system $\mathcal{S}_{loc}$ is then obtained by adding to the basic proof rules the local induction rule (Ind-L) of Table 2. The table also shows its derivable dual (Ind-R), which might look more familiar to the reader.

Given a set $\mathcal{S}$ of proof rules an *$\mathcal{S}$-derivation tree* $\mathcal{D} = (\mathcal{N}, \mathcal{E}, \mathcal{L})$ is a tree $(\mathcal{N}, \mathcal{E})$ with nodes $\mathcal{N}$ and edges $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ together with a function $\mathcal{L}$ labelling each node of the tree with a sequent in a way that is consistent with the application of the proof rules in $\mathcal{S}$. We often write $N(\Gamma \vdash_{\mathcal{O}} \Delta)$ for $\mathcal{L}(N) = \Gamma \vdash_{\mathcal{O}} \Delta$.

Definition 3. *An* $\mathcal{S}_{loc}$-proof for a sequent $\Gamma \vdash_{\mathcal{O}} \Delta$ *is an* $\mathcal{S}_{loc}$*-derivation tree* $\mathcal{D}$ *whose root is labelled by* $\Gamma \vdash_{\mathcal{O}} \Delta$ *and each leaf of which is labelled by an axiom.*

Lemma 1. *The proof rules of Tables 1 and 2 are sound. In particular, if there is a* Σ*-model* $(\mathcal{A}, \rho)$ *falsifying the conclusion* C *of a rule then there is a environment* ρ' *such that* $(\mathcal{A}, \rho')$ *falsifies some premise* P *of that rule. Moreover, we can choose* ρ' *such that, for all ordinal variables* κ *free in both* P *and* C*, we have* $\rho'(\kappa) = \rho(\theta(\kappa))$ *for rule (Subst) and* $\rho'(\kappa) \leq \rho(\kappa)$ *for all other rules.*

The proof proceeds by a straightforward inspection of the rules. In particular, the soundness of the fixed point rules follows immediately from Proposition 1. The soundness of the proof system $\mathcal{S}_{loc}$ is then an immediate corollary.

Theorem 1. (Soundness of $\mathcal{S}_{loc}$) *If* $\mathcal{D}$ *is an* $\mathcal{S}_{loc}$*-proof for* $\Gamma \vdash_{\mathcal{O}} \Delta$ *then* $\Gamma \vdash_{\mathcal{O}} \Delta$ *is valid.*

4 Global Induction: The System $\mathcal{S}_{glob}$

The proof system $\mathcal{S}_{glob}$ uses the proof rules from Table 1 only. However, proofs in this system are not finite but ω-regular trees, which we represent as finite trees with back edges called *repeats*. An external global *induction discharge condition* then ensures that all inductive reasoning embodied in the proof structure is well-founded. Let us fix an arbitrary $\mathcal{S}_{glob}$-derivation tree $\mathcal{D} = (\mathcal{N}, \mathcal{E}, \mathcal{L})$.

Definition 4. (Repeat) *A* repeat $R = (N, M)$ *for* $\mathcal{D}$ *is a pair of nodes of* $\mathcal{D}$ *such that* N *is a leaf,* M *lies on the path from the root of* $\mathcal{D}$ *to* N *and* $\mathcal{L}(N) = \mathcal{L}(M)$*. The node* N *is called the* repeat node *and* M *is called its* companion*. We denote by* $\pi(R)$ *the path* $M \cdots N$ *in* $\mathcal{D}$*.*

Definition 5. (Pre-Proof) *A* pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ *is composed of an* $\mathcal{S}_{glob}$*-derivation tree* $\mathcal{D} = (\mathcal{N}, \mathcal{E}, \mathcal{L})$ *and a set of repeats* $\mathcal{R}$ *for* $\mathcal{D}$ *such that each of its non-axiom leaves appears in exactly one repeat of* $\mathcal{R}$*. We call the graph* $\mathcal{G}(\mathcal{P}) = (\mathcal{N}, \mathcal{E} \cup \mathcal{R}, \mathcal{L})$ *the* pre-proof graph *of* $\mathcal{P}$*.*

The following lemma will allow us to identify strongly connected subgraphs of $\mathcal{G}(\mathcal{P})$ with certain subsets of $\mathcal{R}$. For two repeats $R = (N, M)$ and $R' = (N', M')$ in $\mathcal{R}$ define $R \to R'$ if there is a path $M \cdots N'$ from the companion node M of R to the repeat node N' of R' in the derivation tree $\mathcal{D}$.

Lemma 2. *There is a bijection between the non-trivial strongly connected subgraphs of* $\mathcal{G}(\mathcal{P})$ *and the strongly connected subgraphs of* $(\mathcal{R}, \to)$*.*

We are now ready to define the *basic induction discharge condition* qualifying a pre-proof as a proof. This condition is based on the notions of preservation and progress of repeats with respect to approximant variables.

Definition 6. (Progress, Preservation) *Constraint* $\kappa' < \kappa$ *is called* derivable at $N(\Gamma \vdash_{\mathcal{O}} \Delta)$*, written* $N \vdash \kappa' < \kappa$*, if there is a repeat-free* $\mathcal{S}_{glob}$*-pre-proof for* $\Gamma \vdash_{\mathcal{O}} \kappa' < \kappa, \Delta$*. Let* R *be a repeat with* $\pi(R) = N_0 \cdots N_m$ *and* $N_i(\Gamma_i \vdash_{\mathcal{O}_i} \Delta_i)$ *and let* κ *be an ordinal variable. Then we say*

- R preserves κ*, if* $\kappa \in \mathcal{O}_i$ *for all* i*, and if either* $N_j \vdash \theta(\kappa) < \kappa$ *or* $\theta(\kappa) = \kappa$ *whenever rule (Subst) is applied with* θ *at* N_j*, and*
- R progresses on κ*, if* R *preserves* κ *and rule (Subst) is applied with some* θ *at some* N_j *such that* $N_j \vdash \theta(\kappa) < \kappa$*.*

Definition 7. ($\mathcal{S}_{glob}$-Proof) *A pre-proof* $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ *is an* $\mathcal{S}_{glob}$-proof *if for each strongly connected subgraph* $S \subseteq \mathcal{R}$ *there is an ordinal variable* κ *such that*

1. *some repeat* $R \in S$ *progresses on* κ*, and*
2. *each repeat* $R' \in S$ *preserves* κ*.*

Theorem 2. (Soundness of $\mathcal{S}_{glob}$) *If $\mathcal{P}$ is an $\mathcal{S}_{glob}$-proof for $\Gamma \vdash_{\mathcal{O}} \Delta$ then $\Gamma \vdash_{\mathcal{O}} \Delta$ is valid.*

Proof. (Sketch) By contradiction. Using Lemma 1 we construct an infinite sequence $(N_0, \rho_0) \cdots (N_i, \rho_i) \cdots$ of pairs of nodes and valuations such that each ρ_i falsifies $\mathcal{L}(N_i)$. By the definition of $\mathcal{S}_{glob}$-proof there is an ordinal variable κ such that the sequence $\{\rho_i(\kappa)\}_i$ of ordinals decreases infinitely often from some point on, contradicting the well-foundedness of the ordinals. □

Discharge Using Induction Orders The basic induction discharge mechanism does not exhibit sufficient structure for our purpose of comparing the two systems $\mathcal{S}_{loc}$ and $\mathcal{S}_{glob}$. For this reason we introduce an alternative induction discharge condition, first proposed by Schöpp [10] and generalising the one in [4], which orders the set of repeats appearing in a pre-proof. The new condition turns out to be equivalent to the original one.

Definition 8. (Induction Orders) *A partial order $(\mathcal{R}, \preceq)$ on the set of repeats is called an* induction order *for $\mathcal{P}$, if it is forest-like and every strongly connected subgraph $S \subseteq \mathcal{R}$ has a $\preceq$-greatest element.*

A labelled induction order *$(\mathcal{R}, \preceq, \delta)$ is an induction order $(\mathcal{R}, \preceq)$ together with a map δ assigning an ordinal variable κ to each repeat $R \in \mathcal{R}$. The ordinal variable $\delta(R)$, also written δ_R, is called the* induction variable for R.

The restriction to forest-like partial orders in this definition is adopted here for convenience. It is not necessary for soundness, but sufficient for completeness (see Proposition 2 below).

Definition 9. (Alternative Discharge) *We say that a labelled induction order $(\mathcal{R}, \preceq, \delta)$* discharges *a pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ if for all $R \in \mathcal{R}$*

1. *R progresses on δ_R, and*
2. *R preserves $\delta_{R'}$ whenever $R \preceq R'$.*

Proposition 2. *For any pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ the following are equivalent:*

(i) there is a labelled induction order $(\mathcal{R}, \preceq, \delta)$ discharging $\mathcal{P}$, and
(ii) $\mathcal{P}$ is an $\mathcal{S}_{glob}$-proof.

5 From Local to Global Induction

The translation of $\mathcal{S}_{loc}$-proofs to $\mathcal{S}_{glob}$-proofs is achieved by showing that the local induction rule (Ind-L) is derivable in $\mathcal{S}_{glob}$, in the strong sense that any application of (Ind-L) inside an $\mathcal{S}_{glob}$-proof can be replaced by an equivalent $\mathcal{S}_{glob}$-derivation.

Theorem 3. *The local induction rule* (Ind-L) *is derivable in $\mathcal{S}_{glob}$.*

Proof. Consider the following derivation (omitting two applications of the weakening rule):

$$\dfrac{\dfrac{\dfrac{\Gamma, \exists\kappa.\phi \vdash_{\mathcal{O}} \Delta}{\lceil \Gamma, \phi \vdash_{\mathcal{O},\kappa} \Delta \rceil}\,(\exists\kappa\text{-L})}{\Gamma, \phi \vdash_{\mathcal{O},\kappa} \exists\kappa' < \kappa.\phi[\kappa'/\kappa], \Delta \quad \dfrac{\Gamma, \exists\kappa' < \kappa.\phi[\kappa'/\kappa] \vdash_{\mathcal{O},\kappa} \Delta}{\dfrac{\Gamma, \kappa' < \kappa, \phi[\kappa'/\kappa] \vdash_{\mathcal{O},\kappa,\kappa'} \Delta}{\lfloor \Gamma, \phi \vdash_{\mathcal{O},\kappa} \Delta \rfloor}\,(\text{Subst})}\,(\exists\kappa\text{-L},\wedge\text{-L})}\,(\text{Cut})$$

This derivation is sound provided $\kappa \notin \mathcal{O}$, the side condition of rule (Ind-L). Since the repeat (indicated by brackets) preserves all variables in $\mathcal{O}$ and progresses on κ, this derivation can safely replace applications of (Ind-L) in $\mathcal{S}_{glob}$-proofs. □

6 From Global to Local Induction

In general, the translation from $\mathcal{S}_{glob}$-proofs to $\mathcal{S}_{loc}$-proofs proceeds in two stages. If the inductive structure of the $\mathcal{S}_{glob}$-proof matches its tree structure, it can be translated directly into an $\mathcal{S}_{loc}$-proof. Otherwise, the $\mathcal{S}_{glob}$-proof needs to be unfolded prior to this transformation. We fix an arbitrary pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{R})$.

Definition 10. (Structural dependency) *The* structural dependency relation $\leq_{\mathcal{P}}$ *on* $\mathcal{R}$ *is defined as follows:* $R' \leq_{\mathcal{P}} R$ *holds for two repeats* $R, R' \in \mathcal{R}$ *if the companion node of* R' *appears on the path* $\pi(R)$.

Lemma 3. *Let* $S \subseteq \mathcal{R}$. *Then* S *is strongly* $\rightarrow$*-connected if and only if* S *is weakly* $\leq_{\mathcal{P}}$*-connected.*

Definition 11. (Tree-compatibility) *An induction order* $(\mathcal{R}, \preceq)$ *for* $\mathcal{P}$ *is* tree-compatible *if* $R \leq_{\mathcal{P}} R'$ *and* $R' \not\leq_{\mathcal{P}} R$ *imply* $R \preceq R'$ *for all* $R, R' \in \mathcal{R}$. *An* $\mathcal{S}_{glob}$*-proof* $\mathcal{P}$ *is called* tree-dischargeable *if there is a tree-compatible induction order discharging* $\mathcal{P}$.

The inductive structure of a tree-dischargeable proof matches its underlying tree structure. The next lemma, which can be proved using Lemma 3, gives a useful characterisation of induction orders for $\mathcal{P}$ in terms of the structural dependency relation $\leq_{\mathcal{P}}$, thereby relating the structure of the proof tree with the dependencies between repeats in an arbitrary induction order.

Lemma 4. *Let* $(\mathcal{R}, \preceq)$ *be a forest-like partial order. Then* $(\mathcal{R}, \preceq)$ *is an induction order for* $\mathcal{P}$ *if and only if* $R \leq_{\mathcal{P}} R'$ *implies* $R \preceq R'$ *or* $R' \preceq R$ *for all* $R, R' \in \mathcal{R}$.

Let $\preceq_{\mathcal{P}}$ be the transitive closure of $\leq_{\mathcal{P}}$. The following two remarks are easy corollaries of Lemma 4.

Proposition 3. *Let $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ be a pre-proof such that $\mathcal{R}$ is* injective *(as a function from repeat nodes to companions). Then $\preceq_{\mathcal{P}}$ is a tree-compatible induction order for $\mathcal{P}$.*

Lemma 5. *Let $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ be a proof discharged by the tree-compatible induction order $(\mathcal{R}, \preceq, \delta)$. Then $\mathcal{P}$ can be transformed into a proof $\mathcal{P}' = (\mathcal{D}', \mathcal{R}')$ of the same sequent such that $\mathcal{R}'$ is injective and $\mathcal{P}'$ is discharged by $(\mathcal{R}', \preceq_{\mathcal{P}'}, \delta')$ for some labelling δ'.*

6.1 Translating Tree-Dischargeable Proofs

Since each repeat R embodies an induction progressing on variable δ_R along the path $\pi(R)$ from the companion to the repeat node, it seems natural to insert the local induction rule (Ind-L) at the companion node and use, essentially, the whole sequent as an induction hypothesis. This induction hypothesis is then conveyed down the proof tree, exploiting progress on δ_R along $\pi(R)$ to remove the bounded quantification introduced by (Ind-L) and thus making the induction hypothesis available for local discharge at the repeat node. The rest of this section is devoted to proving the following theorem along these lines.

Theorem 4. *Let $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ be a $\mathcal{S}_{glob}$-proof of $\Gamma \vdash_{\mathcal{O}} \Delta$, tree-dischargeable by $(\mathcal{R}, \preceq, \delta)$. Then $\mathcal{P}$ can be transformed into a $\mathcal{S}_{loc}$-proof $\mathcal{D}'$ of $\Gamma \vdash_{\mathcal{O}} \Delta$.*

Proof. (Sketch) We assume w.l.o.g. that (A) $\mathcal{R}$ is injective and $\preceq = \preceq_{\mathcal{P}}$, by Lemma 5, and (B) companions appear only as descendents of nodes where a rule other than (Subst), (Cut), ($\wedge$-R) and ($<$-R) is applied.

Our construction recursively transforms $\mathcal{D} = (\mathcal{N}, \mathcal{E}, \mathcal{L})$ into a new derivation tree $\mathcal{D}' = (\mathcal{N}', \mathcal{E}', \mathcal{L}')$ by replacing at each node N the rule applied to produce the set of descendents $\mathcal{N}_N$ by a derivation $\mathcal{D}_N$ with root $\widehat{N}$ and premises $\widehat{\mathcal{N}_N} = \{\widehat{N'} \mid N' \in \mathcal{N}_N\}$ (and some fresh interior nodes). The procedure keeps the set $\mathcal{H}$ of current *induction hypotheses*, which is added to $N(\Gamma \vdash_{\mathcal{O}} \Delta)$ yielding $\widehat{N}(\Gamma \vdash_{\mathcal{O}} \Delta, \mathcal{H})$ in $\mathcal{D}'$. For any node $N \in \mathcal{N}$ define the set of repeats *active* at N by $\mathcal{R}_N = \{R' \in \mathcal{R} \mid \exists R \in \mathcal{R}. N \in \pi(R) \text{ and } R \preceq_{\mathcal{P}} R'\}$ and let $\mathcal{O}_N = \{\delta_R \mid R \in \mathcal{R}_N\}$ be the corresponding set of induction variables. We note fact (C): there is some $R \in \mathcal{R}_N$ preserving all variables in $\mathcal{O}_N$. Hence, $\mathcal{O}_N \subseteq \mathcal{O}$ for $N(\Gamma \vdash_{\mathcal{O}} \Delta)$.

The induction hypothesis H_R to be added to the current set of hypotheses $\mathcal{H}$ at the companion node $M(\Gamma \vdash_{\mathcal{O}} \Delta)$ of a repeat $R = (N, M)$ is defined by

$$\begin{aligned} H_R &= \exists \delta < \delta_R. \Psi_R[\delta/\delta_R] \\ \Psi_R &= \exists \overline{\iota} \leq \overline{\iota}_R. \exists \overline{v}. \bigwedge (\Gamma \cup \neg\Delta \cup \neg\mathcal{H})[\overline{\iota}/\overline{\iota}_R] \end{aligned}$$

where $\{\overline{\iota}_R\} = \mathcal{O}_M - \{\delta_R\}$ and $\{\overline{v}\} = \mathrm{fv}(\Gamma \vdash_{\mathcal{O}} \Delta) - \mathcal{O}_M$. The *free* induction hypothesis Ψ_R packs the sequent at M together with the set $\mathcal{H}$ into a single formula, existentially quantifies all but the active induction variables in $\mathcal{O}_M$ and binds the (preserved) active induction variables $\overline{\iota}_R$ in Ψ_R by a quantifier

of type $\exists \cdot \leq \overline{\tau}_R$. The *guarded* induction hypothesis H_R additionally binds the (progressing) induction variable δ_R in Ψ_R by a quantifier of type $\exists \cdot \leq \delta_R$.

Transformation of $\mathcal{D}$ to $\mathcal{D}'$. Our procedure ensures that for each node $N(\Gamma \vdash_{\mathcal{O}} \Delta)$ in $\mathcal{D}$ there is a node $\widehat{N}(\Gamma \vdash_{\mathcal{O}} \Delta, \mathcal{H})$ such that the following invariant holds:

$$I(N, \mathcal{H}) = \ \mathcal{H}^g_N \subseteq \mathcal{H} \subseteq \mathcal{H}_N$$

where $\mathcal{H}^g_N = \{H_R \mid R \in \mathcal{R}_N\}$ and $\mathcal{H}_N = \mathcal{H}^g_N \cup \{\Psi_R \mid R \in \mathcal{R}_N\}$. By assumption (B) the root N_r is not a companion so $\mathcal{R}_N = \varnothing$ and the invariant holds trivially by initially setting $\mathcal{H} = \varnothing$. We now describe the derivations $\mathcal{D}_N$ replacing in $\mathcal{D}'$ the rule application at N in $\mathcal{D}$. Suppose we have constructed $\mathcal{D}'$ up to some node $\widehat{N}(\Gamma' \vdash_{\mathcal{O}'} \Delta', \mathcal{H})$ where the invariant holds. The cases where a rule other than (Subst) is applied at N and none of the descendents of N is a companion are easy to show. We just remark that in order to maintain the invariant at the branching rules (Cut), ($\wedge$-R) and ($<$-R) we possibly need weakening on $\mathcal{H}$ to account for the splitting of the set of active repeats between the two branches. Due to assumption (B) the two remaining cases are:

Case 1. The single descendent $M(\Gamma \vdash_{\mathcal{O}} \Delta)$ of N is the companion of a repeat R. The invariant $I(M, \mathcal{H})$ is violated for $\mathcal{H}$, because the induction hypothesis H_R is missing in $\mathcal{H}$. The derivation $\mathcal{D}_N$ in Fig. 1 adds H_R to $\widehat{N}(\Gamma' \vdash_{\mathcal{O}'} \Delta', \mathcal{H})$ yielding $\widehat{M}(\Gamma \vdash_{\mathcal{O}} \Delta, \mathcal{H}, H_R)$ and thus reestablishing the invariant.

At node M' we cut in $\exists \delta_R . \Psi_R$. After weakening away all but the latter formula on the left hand side, we apply the induction rule (Ind-L). This leaves us with the sequent $\Psi_R \vdash_{\mathcal{O}_N} H_R$, which is transformed into the desired sequent $\widehat{M}(\Gamma \vdash_{\mathcal{O}} \Delta, \mathcal{H}, H_R)$ by applying a series of essentially first-order rules (RS1) deconstructing Ψ_R on the left. On the right hand side, we apply a dual series of rules (RS2) proving $N_r(\Gamma \vdash_{\mathcal{O}} \Delta, \mathcal{H}, \Psi_R)$ locally. Note that the sequent at N_r anticipates, thanks to the cut, the desired situation at the repeat node of R, since it contains a right hand side occurrence of the free induction hypothesis.

$$\dfrac{\dfrac{\dfrac{\dfrac{\widehat{N}(\Gamma' \vdash_{\mathcal{O}'} \Delta', \mathcal{H})}{M'(\Gamma \vdash_{\mathcal{O}} \Delta, \mathcal{H})}\,(\text{rule at } N)}{\dfrac{\Gamma, \exists \delta_R . \Psi_R \vdash_{\mathcal{O}} \Delta, \mathcal{H}}{\exists \delta_R . \Psi_R \vdash_{\mathcal{O}_N - \{\delta_R\}}}\,(\text{Weak})}\,(\text{Ind-L})}{\Psi_R \vdash_{\mathcal{O}_N} H_R}\,(\text{RS1})}{\widehat{M}(\Gamma \vdash_{\mathcal{O}} \Delta, \mathcal{H}, H_R)}\,(\text{Cut}) \qquad \dfrac{\dfrac{\Gamma \vdash_{\mathcal{O}} \Delta, \mathcal{H}, \exists \delta_R . \Psi_R}{N_r(\Gamma \vdash_{\mathcal{O}} \Delta, \mathcal{H}, \Psi_R)}\,(\text{RS2})}{\cdot}\,(\exists\kappa\text{-R})$$

Fig. 1. Derivation $\mathcal{D}_N$ inserted at companion node N of a repeat R

Case 2. Rule (Subst) applied at N with descendent M. We need to make sure that the substitution rule is correctly applied and that the induction hypotheses in $\mathcal{H}$ are (re-)generated as in the following compressed version of derivation $\mathcal{D}_N$:

$$\dfrac{\dfrac{\widehat{N}(\Gamma[\theta] \vdash_{\mathcal{O}[\theta]} \Delta[\theta], \mathcal{H})}{M'(\Gamma[\theta] \vdash_{\mathcal{O}[\theta]} \Delta[\theta], \mathcal{H}'[\theta])}\,(\text{Regen})}{\widehat{M}(\Gamma \vdash_{\mathcal{O}} \Delta, \mathcal{H}')}\,(\text{Subst})$$

where $\mathcal{H}' = \{H_R \mid H_R \in \mathcal{H}\} \cup \{\Psi_R \mid \Psi_R \in \mathcal{H} \text{ or } N \vdash \theta(\delta_R) < \delta_R\}$. We have $\mathcal{R}_M = \mathcal{R}_N$, since M is not a companion by assumption (B). It is then easy to see that $I(M, \mathcal{H}')$ holds. Note that, since $\mathrm{fv}(\mathcal{H}_N) \subseteq \mathcal{O}_N$, it follows from fact (C) above that each $\kappa \in \mathrm{fv}(\mathcal{H}_N)$ is preserved by θ at N, that is, $N \vdash \theta(\kappa) < \kappa$ or $\theta(\kappa) = \kappa$. The derivation from $\widehat{N}$ to M' labelled (Regen) then includes, for each $R \in \mathcal{R}_N$, a derivation that produces:

1. $\Psi_R[\theta]$ from H_R if $N \vdash \theta(\delta_R) < \delta_R$, exploiting progress of δ_R and preservation of $\overline{\tau}_R$ by θ at N,
2. $\Psi_R[\theta]$ from Ψ_R if $\theta(\delta_R) = \delta_R$ and $\Psi_R \in \mathcal{H}$, using preservation of δ_R and $\overline{\tau}_R$ by θ at N, and
3. $H_R[\theta]$ from H_R, also using preservation of δ_R and $\overline{\tau}_R$ by θ at N.

In each of the derivations (1)-(3) the leading bounded ordinal quantifiers in Ψ_R and H_R are duplicated and commuted prior to instantiation as necessary, by applying some easily derivable auxiliary rules.

Continuing this procedure down to the leaves of $\mathcal{D}$ yields, for each repeat $R = (N, M)$, two nodes $\widehat{M}(\Gamma \vdash_{\mathcal{O}} \Delta, \mathcal{H})$ and $\widehat{N}(\Gamma \vdash_{\mathcal{O}} \Delta, \mathcal{H}')$ in $\mathcal{D}'$ (where $\widehat{N}$ is a leaf of $\mathcal{D}'$ so far). We now show that $\mathcal{H} \subseteq \mathcal{H}'$ and $\Psi_R \in \mathcal{H}'$, implying that the sequent at $\widehat{N}$ is provable in $\mathcal{S}_{loc}$ in the same way as the one at node N_r in Fig. 1. Consider some $R' \in \mathcal{R}_M$. From $\mathcal{R}_M = \mathcal{R}_N$ and the invariant it follows that $H_{R'}$ is in both $\mathcal{H}_M$ and $\mathcal{H}_N$. If $\Psi_{R'} \in \mathcal{H}_M$ then we also have $\Psi_{R'} \in \mathcal{H}_N$, since R preserves $\delta_{R'}$ and so $\Psi_{R'}$ is regenerated along $\pi(R)$ (see discussion in case (2) above). Hence, $\mathcal{H} \subseteq \mathcal{H}'$. Since R progresses on δ_R, Ψ_R will be generated from H_R at some point on the path $\pi(R)$ and then regenerated in each subsequent application of rule (Subst) along $\pi(R)$. Hence, we have $\Psi_R \in \mathcal{H}'$. This shows that $\mathcal{D}'$ can be completed into an $\mathcal{S}_{loc}$-proof. □

6.2 General Case: Unfolding Proofs

The previous translation crucially depends on the tree-dischargeability of the induction order: repeats with companions lower in the proof tree preserve induction variables of repeats higher in the proof tree ("higher" and "lower" being determined by $\preceq_{\mathcal{P}}$). In general, we need to unfold the proof until it becomes tree-dischargeable. This task is achieved by Algorithm 1.

```
 1: input
 2:    P_0 = (D_0, R_0) where D_0 = (N_0, E_0, L_0), root N_r  { R_0 injective }
 3:    (R_0, ≺_0, δ_0)  { induction order discharging P_0 }
 4: output
 5:    P = (D, R) where D = (N, E, L), N ⊆ N_0 × ℕ  { unfolded proof }
 6: globals
 7:    s ∈ ℕ  { sequence number to distinguish copies of nodes }
 8: begin
 9:    s := 0; E := ∅; R := ∅
10:    N := {(N_r, s)}; L := {((N_r, s), L_0(N_r))}
11:    unfold (N_r, s) ∅
12: end

13: procedure unfold (N, k) B
14: parameters
15:    (N, k) ∈ N_0 × ℕ  { node of P = (node of P_0, copy number) }
16:    B: R_0 ⇀ ℕ  { copy numbers for companions available in P }
17: if N is the repeat node of some R = (N, M) ∈ R_0  then
18:    if (R, i) ∈ B for some i  then { companion (M, i) available for (N, k) }
19:       R := R ∪ {((N, k), (M, i))}
20:    else { no companion available, continue unfolding through repeat }
21:       s := s + 1  { get a new sequence number }
22:       add node (M, s) labelled L_0(M) as child of (N, k) to D
23:       unfold (M, s) B
24:    end if
25: else { N is an axiom leaf or an inner node of D_0 }
26:    if N is the companion of some R ∈ R_0 and R ∉ dom B  then
27:       B := {(R, k)} ∪ {(R', k') ∈ B |  R ≺_0 R'}
28:    end if
29:    for each child N' of N in D_0  do { add and unfold each child node }
30:       add node (N', k) labelled L_0(N') as child of (N, k) to D
31:       unfold (N', k) B
32:    end for
33: end if
```

Algorithm 1. Unfolding proofs

It takes a proof $\mathcal{P}_0 = (\mathcal{D}_0, \mathcal{R}_0)$ with injective $\mathcal{R}_0$ as input and produces a tree-dischargeable proof $\mathcal{P} = (\mathcal{D}, \mathcal{R})$. Note that no generality is lost by requiring that $\mathcal{R}_0$ is injective. The nodes of $\mathcal{P}$ are pairs (N, k), where N is a node of $\mathcal{P}_0$ and k is the copy number. The original proof tree is traversed recursively, unfolding repeats into new copies of the proof tree as necessary. The procedure maintains a partial map $\mathcal{B}$ from repeats $\mathcal{R}_0$ to copy numbers in $\mathbb{N}$ to keep track of companions that are available for looping back at repeat nodes. This map is updated whenever we encounter the companion of some repeat $R \in \mathcal{R}$ without an entry in $\mathcal{B}$: the entry (R, k) is added to $\mathcal{B}$, while any entry for a repeat R' not above R with respect to $\prec_0$ is removed from $\mathcal{B}$ (line 26-27). When examining

copy l of the repeat node N of some repeat $R = (N, M) \in \mathcal{R}$ we check whether there is some entry $(R, k) \in \mathcal{B}$ (lines 17-18). If so, then we can safely close the loop and add $((N, l), (M, k))$ as a new repeat to $\mathcal{R}$ (line 19). Otherwise, if there is no companion available for R, we proceed by unfolding the tree at (N, l) by adding the node (M, k) with a fresh k as a descendant of (N, l) to $\mathcal{P}$ (line 21-23).

The labelled induction order $(\mathcal{R}, \preceq, \delta)$ for $\mathcal{P}$ is obtained by lifting $(\mathcal{R}_0, \preceq_0, \delta_0)$ to $\mathcal{P}$. Writing $\widehat{R} = (N, M) \in \mathcal{R}_0$ for $R = ((N, k), (M, i)) \in \mathcal{R}$, we define

$$R \preceq R' \Leftrightarrow \widehat{R} \prec_0 \widehat{R'} \text{ or } (\widehat{R} = \widehat{R'} \text{ and } k \leq k') \quad \text{and} \quad \delta(R) = \delta_0(\widehat{R})$$

where $R = ((N, k), (M, i))$ and $R' = ((N', k'), (M', i'))$. Note the tie-breaking role of the repeat sequence number in case of identical projected repeats.

Theorem 5. *Let $\mathcal{P}_0 = (\mathcal{D}_0, \mathcal{R}_0)$ be a $\mathcal{S}_{glob}$-proof of $\Gamma \vdash_{\mathcal{O}} \Delta$ such that $\mathcal{R}_0$ is injective and $\mathcal{P}_0$ is discharged by $(\mathcal{R}_0, \preceq_0, \delta_0)$. Then Algorithm 1 produces, in time $\mathcal{O}(2^{|\mathcal{N}_0| \times |\mathcal{R}_0|})$, a proof $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ of $\Gamma \vdash_{\mathcal{O}} \Delta$, discharged by the tree-compatible induction order $(\mathcal{R}, \preceq, \delta)$ defined above.*

Proof. For partial correctness it is sufficient to show that

1. $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ is a pre-proof,
2. $(\mathcal{R}, \preceq)$ is a tree-compatible induction order for $\mathcal{P}$, and
3. $(\mathcal{R}, \preceq, \delta)$ discharges $\mathcal{P}$.

It is easy to see that $\mathcal{P}$ is a pre-proof. As a preparation for (2) and (3) consider a repeat $R = ((N, k), (M, i))$ in $\mathcal{R}$. Let

$$c = ((N_0, k_0), \mathcal{B}_0) \cdots ((N_j, k_j), \mathcal{B}_j) \cdots ((N_m, k_m), \mathcal{B}_m)$$

with $(N_0, k_0) = (M, i)$ and $(N_m, k_m) = (N, k)$ be the (suffix of the) sequence of recursive calls leading to the introduction of R in $\mathcal{P}$. The pair $(\widehat{R}, i)$ is added to $\mathcal{B}_0$ in call c_0 (line 27) and appears in all subsequent $\mathcal{B}_j$ $(1 \leq j \leq m)$. Finally, the repeat R is added to $\mathcal{R}$ in call c_m (line 19). Consider a repeat $R'' = (N'', M'') \in \mathcal{R}_0$ with its companion $M'' = N_j$ occurring in c_j with $0 \leq j \leq m$. Since $(\widehat{R}, i)$ is in both $\mathcal{B}_j$ and $\mathcal{B}_{j+1}$ and (R'', k_j) is added to $\mathcal{B}_j$ at in call c_j (line 27), we have

(A) if $R'' \notin \operatorname{dom} \mathcal{B}_j$ then $R'' \preceq_0 \widehat{R}$.

Ad (2). It is easy to check that $(\mathcal{R}, \preceq, \delta)$ is a forest-like partial order. Now suppose $R' \leq_{\mathcal{P}} R$ for some $R, R' \in \mathcal{R}$. Since the companion of R' appears on $\pi(R)$ in $\mathcal{P}$ and hence as N_j in some c_j it must be the case that $\widehat{R'} \notin \operatorname{dom} \mathcal{B}_j$ so we have $\widehat{R'} \preceq_0 \widehat{R}$ by (A). From the definition of $\preceq$ and Lemma 4 it follows that $(\mathcal{R}, \preceq)$ is an induction order. If, moreover, $R \not\leq_{\mathcal{P}} R'$ then $\widehat{R'} \neq \widehat{R}$, since $\widehat{R}$ cannot be unfolded on $\pi(R)$. This implies that $(\mathcal{R}, \preceq)$ is tree-compatible.

Ad (3). Let $S' \subseteq \mathcal{R}_0$ be the set of repeats unfolded on $\pi(R)$ and let $S = S' \cup \{\widehat{R}\}$. Note that $\pi(R)$ in $\mathcal{P}$ is the composition of all the paths $\pi(R'')$ in $\mathcal{P}_0$ with $R'' \in S$.

Suppose $R \preceq R'$ for some $R' \in \mathcal{R}$. Then certainly $\widehat{R} \preceq_0 \widehat{R'}$. Let $R'' \in S'$ with companion M. Since R'' is unfolded in some call c_{j-1}, we have $N_j = M$ and $R'' \notin \operatorname{dom} \mathcal{B}_j$. Hence, $R'' \preceq_0 \widehat{R}$ by (A) and R'' preserves $\delta(R') = \delta_0(\widehat{R'})$, implying that R preserves $\delta(R')$. Moreover, R progresses on $\delta(R)$, since $\widehat{R}$ progresses on $\delta_0(\widehat{R})$. This shows (3).

Complexity. Suppose for a contradiction that there is a (suffix of a) sequence of calls of the form c above such that $m > 0$ and $(N_0, \operatorname{dom} \mathcal{B}_0) = (N_m, \operatorname{dom} \mathcal{B}_m)$. Note first that, since the control flow of `unfold` does not depend on the copy numbers, there is an extension $((N_{m+1}, k_{m+1}), \mathcal{B}_{m+1}) \cdots ((N_{2m}, k_{2m}), \mathcal{B}_{2m})$ of c such that $(N_i, \operatorname{dom} \mathcal{B}_i) = (N_{i+m}, \operatorname{dom} \mathcal{B}_{i+m})$ for all $i \leq m$ (and, in fact, so on ad infinitum). Let S be the set of repeats $R \in \mathcal{R}_0$ such that N_i is the companion of R and $R \notin \operatorname{dom} \mathcal{B}_i$ for some $i \leq m$. It is not difficult to see that S is strongly connected in $(\mathcal{R}_0, \rightarrow)$ and thus there is a $\preceq_0$-greatest element $\widetilde{R} \in S$. Suppose N_i is the companion of $\widetilde{R}$ for some $i \leq m$. Since $\widetilde{R} \notin \operatorname{dom} \mathcal{B}_i$ and $\widetilde{R}$ is $\preceq_0$-greatest in S, it follows that $\widetilde{R} \in \operatorname{dom} \mathcal{B}_k$ for all $i < k \leq i + m$ (line 27). In particular, $\widetilde{R} \in \operatorname{dom} \mathcal{B}_{i+m}$, which contradicts $\operatorname{dom} B_i = \operatorname{dom} B_{i+m}$. Hence, length of any call sequence c of `unfold` is bounded by $|c| \leq |\mathcal{N}_0| \times |\mathcal{R}_0|$, yielding an upper bound of $|\mathcal{N}| \leq 2^{|\mathcal{N}_0| \times |\mathcal{R}_0|}$ for the size of $\mathcal{P}$ and the time complexity of the algorithm. □

7 Conclusions

We have presented a translation between proofs using well-founded induction and cyclic proofs based on a global well-foundedness condition. The proof systems use explicit ordinal approximations as suggested in [4]. Since our main interest in approximants is as a proof-theoretical mechanism to deal with fixed points rather than proving theorems about them, it would be desirable to identify a fragment of the language which could be shown to conserve (not increase) the expressiveness of the basic μ-calculus (without explicit approximations). Simpson and Schöpp have proposed an alternative approach to approximants based directly on the second-order variables instead of ordinal variables [11] and they have proved a conservativity result for a variant of this language [12]. Their language lacks, however, the ability to "internalise" sequents into single formulas, required in our direct translation to local proofs. Thus, it seems that our approach can not readily be adapted to yield a similar translation in their framework.

On a different line of research, we would like to investigate whether the ideas of this paper can be transferred to the context of type theories with inductive definitions such as the Calculus of Inductive Constructions [9]. A useful starting point is the introduction of approximated inductive types along the lines of [2].

Acknowledgements We would like to thank Alex Simpson and Uli Schöpp as well as the members of the FDT group at SICS for fruitful discussions on the topic. We are also grateful to Dilian Gurov, Marieke Huisman and the anonymous referees for their helpful suggestions.

References

[1] T. Arts, G. Chuganov, M. Dam, L.-å. Fredlund, D. Gurov, and T. Noll. A tool for verifying software written in Erlang. Accepted for publication in *STTT Journal*, 2001.

[2] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 2000. to appear.

[3] H. Comon. Inductionless induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 14. Elsevier Science, 2001.

[4] M. Dam and D. Gurov. μ-calculus with explicit points and approximations. *Journal of Logic and Computation*, 12(2):43–57, 2002. Previously appeared in Fixed Points in Computer Science, FICS '02.

[5] L. Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.

[6] R. Goré. Tableau methods for modal and temporal logics. In *Handbook of Tableau Methods*. Kluwer, 1999.

[7] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[8] D. Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3(2):173–181, 1976.

[9] C. Paulin-Mohring. Inductive definitions in the system Coq – rules and properties. Technical Report 92-49, Laboratoire de l'Informatique du Parallélisme, ENS Lyon, France, Dec. 1992.

[10] U. Schöpp. Formal verification of processes. Master's thesis, University of Edinburgh, 2001.

[11] U. Schöpp and A. Simpson. Verifying temporal properties using explicit approximants: Completeness for context-free processes. In *FOSSACS '02, Grenoble, France*, volume 2303 of *LNCS*, pages 372–386. Springer-Verlag, 2002.

[12] A. Simpson and U. Schöpp. Private communication.

[13] C. Stirling and D. Walker. Local model checking in the modal μ-calculus. *Theoretical Computer Science*, 89:161–177, 1991.

[14] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier Science Publishers, Amsterdam, 1990.

[15] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Logic in Computer Science, LICS '86*, pages 322–331, 1986.

[16] I. Walukiewicz. Completeness of Kozen's axiomatisation of the propositional μ-calculus. In *Logic in Computer Science, LICS '95*, pages 14–24, 1995.

Author Index